Quantum Software Engineering

Landscapes and Horizons

Jianjun Zhao
Kyushu University
zhao@ait.kyushu-u.ac.jp

ABSTRACT

Quantum software plays a critical role in exploiting the full potential of quantum computing systems. As a result, it is drawing increasing attention recently. This paper defines the term "quantum software engineering" and introduces a quantum software life cycle. Based on these, the paper provides a comprehensive survey of the current state of the art in the field and presents the challenges and opportunities that we face. The survey summarizes the technology available in the various phases of the quantum software life cycle, including quantum software requirements analysis, design, implementation, test, and maintenance. It also covers the crucial issue of quantum software reuse.

KEYWORDS

Quantum software engineering, quantum programming, quantum software life cycle, requirements analysis, design, testing and debugging, maintenance, and reuse

1 INTRODUCTION

"The challenge [of quantum software engineering] is to rework and extend the whole of classical software engineering into the quantum domain so that programmers can manipulate quantum programs with the same ease and confidence that they manipulate today's classical programs."

Susan Stepney, in the 2004 report of Grand Challenges in Computing Research [130].

Quantum computing is a rapidly developing field that is expected to make breakthroughs in many areas [20, 25, 67, 161, 204, 205]. Quantum computing uses the principles of quantum mechanics to process information and has the potential to perform specific tasks much faster than classical computing. The basis of quantum computing is the quantum bit (qubit). Unlike a classical computer bit that is assigned either 0 or 1, a qubit can be either assigned states that is the superposition of 0 and 1. Compared to classical algorithms, quantum algorithms have the potential to solve specific problems with exponential acceleration. In the last two decades, there has been a large amount of literature that contributed to the development of quantum algorithms [70, 118, 192, 193, 249, 257].

Just like classical computing, quantum computing is applicable to applications in many disciplines and will have a wide impact [108]. There are two types of applications where quantum computers are expected to outperform classical computers [178]. The first application is the problems that require large amounts of parallel computing, such as optimization [81, 121], encryption [194], big data analysis [230], and machine learning [34, 76]. The other application is the problems that require efficient and accurate simulation of quantum problems in nature [85, 307] from the areas, such as physics [52], chemistry [182, 204, 231] and materials science [116, 299].

With the rapid development of impressive quantum hardware as well as the accessibility of universal quantum devices to the researchers and professionals via Quantum-as-a-Service (QaaS) [17, 167], it is high time to focus our attention on engineering quantum software systems to reap the benefits of quantum technology. Meanwhile, various application domains of quantum computing [144, 275, 290, 291] urgently need quantum software engineering methodologies and techniques to solve their specific problems. Therefore, we believe that there is an urgent need to build a community for quantum software engineering that focuses on devising methods, tools, and processes for developing quantum software systems efficiently.

Quantum software development techniques aim at providing means for creating quantum software throughout the quantum software life cycle (refer to Section 3.5). A number of quantum programming approaches are available, for instance, Scaffold [1, 138], Qiskit [232], Q# [269], ProjectQ [272], and Quipper[115]. Moreover, the concepts are also being applied at the earlier stages of quantum software development. For example, at the quantum software design stage, [45, 46, 220] provides means for modeling and specifying quantum software systems, and [260] studied how the characteristics of quantum computing systems impact the quality attributes of quantum software architectures. With a variety of techniques available to a software engineer at each stage, the task of engineering a quantum software system poses significant challenges. At each development stage, the software engineer needs to employ the most suitable quantum software technique for the application being developed. The choice of technique can be dictated by various factors, including system requirements, organizational practices, and constraints imposed by the tools or development environments, and the nature of the quantum programming (mechanics). This implies that multiple techniques may be employed at each stage in conjunction with each other. As quantum software development techniques mature, there is a need for guidelines supporting the development of well-engineered quantum software systems.

In this paper, we present a comprehensive survey of quantum software engineering. We summarize the aspects of previous work that have focused mainly on quantum software engineering issues, while simultaneously covering various approaches across different phases of the quantum software life cycle. We have organized the literature according to five different aspects: requirements, design, implementation, testing, and maintenance. Some papers may cover more than one aspect. For such papers, we refer them to all the relevant sections to ensure completeness.

Additionally, we identify challenges and opportunities for the emerging research community working at the intersection between methodologies and techniques for classical software engineering and problems in engineering quantum software systems. To ensure that our survey is self-contained, we have tried to include enough materials to thoroughly guide the software engineering researchers who are interested in techniques and methods for engineering quantum software systems. We also seek to provide quantum computing researchers with a complete survey of software engineering solutions to improve the development of quantum systems.

There have been surveys related to some aspects of quantum software engineering. For example, a number of surveys [54, 98, 102, 125, 140, 186, 239, 244, 261, 263, 276, 280, 281, 302, 311] have been proposed for quantum programming languages, which mainly focus on the language paradigms, implementations, semantics, and tools. Surveys related to quantum software development environments [86, 163, 236, 253] have also been proposed, from various points of view. However, as far as we know, no previous work has focused on a comprehensive survey of the whole life cycle of quantum software development, including quantum software requirements analysis, design, implementation, testing, and maintenance.

In summary, the main contributions of this paper are as follows:

- **Definition.** It defines the term "quantum software engineering" and introduces a quantum software life cycle model for supporting quantum software development.
- **Survey.** It provides a comprehensive survey on quantum software engineering, which is across various phases of quantum software life cycle, including quantum software requirements analysis, design, implementation, testing, and maintenance.
- **Horizons.** It identifies challenges, opportunities, and promising research directions for quantum software engineering, intended to promote and stimulate further research.

The rest of this paper is organized as follows. The structure of this paper is also depicted in Table 1. Section 2 reviews the fundamental terminology in quantum computing. Section 3 briefly introduces the quantum software development which includes quantum programming, a definition of "quantum software engineering," and the quantum software life cycle for quantum software development. From Section 4 to Section 8, we survey the current state of the art on quantum software requirements analysis, design, implementation, testing, and maintenance, respectively. Section 9 covers the current state of the art on quantum software reuse. Section 10 discusses the challenges and opportunities for quantum software engineering. Related works are discussed in Section 11, and concluding remarks are given in Section 12. An extensive set of references is provided for readers wishing to pursue the matter further.

2 BACKGROUND

This section briefly introduces some basics of quantum mechanics, which form the basis of quantum computing. More detailed reading materials can be found in the books by Gruska [120], Nielsen & Chuang [202], and Nakahara [198]. Preskill's lecture notes [225] are also a valuable resource.

2.1 Quantum Bit (Qubit)

A classical bit is a binary unit of information used in classical computation. It can take two possible values, 0 or 1. A quantum bit (or qubit) is different from the classical bit in that its state is theoretically represented by a linear combination of two bases in the quantum state space (represented by a column vector of length 2). We can define two qubits $|0\rangle$ and $|1\rangle$, which can be described as follows:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Qubits $|0\rangle$ and $|1\rangle$ are the computational basis state of the qubit. In other words, they are a set of basis of quantum state space.

Any qubit $|e\rangle$ can be expressed as a linear combination of two basis as $\alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers, and $|\alpha|^2 + |\beta|^2 = 1$. This restriction is also called *normalization conditions*.

For example, $0.6|0\rangle + 0.8|1\rangle$ is a qubit state, and $\frac{1-i}{2}|0\rangle + \frac{1+i}{2}|1\rangle$ is also a qubit state, but $(1-i)|0\rangle + (1+i)|1\rangle$ is not a legitimate qubit state because it does not satisfy the normalization condition. Intuitively, qubits can be viewed as a superposition of classical bits. For example, $0.6|0\rangle + 0.8|1\rangle$ can be considered as a superposition state of $|0\rangle$ and $|1\rangle$, where the probability for bit $|0\rangle$ is $0.6^2 = 0.36$ and for bit $|1\rangle$ is $0.8^2 = 0.64$. Note that $0.6^2 + 0.8^2 = 1$.

2.2 Quantum Gate

Just as a logic gate in a digital circuit that can modify the state of a bit, a quantum gate can change the state of a qubit. A quantum gate can have only one input and one output (transition of a single quantum state), or it can have multiple inputs and multiple outputs (transition of multiple quantum states). The number of inputs and outputs should be equal because the operators need to be reversible which means no information can be lost in quantum computing. Here, we describe two quantum gates with single input and output, and one quantum gate with multiple inputs and outputs.

2.2.1 **NOT Gate.** The NOT gate works on a single qubit. It can exchange the coefficients of two basis vectors:

$$NOT(\alpha|0\rangle + \beta|1\rangle) = \alpha|1\rangle + \beta|0\rangle$$

The quantum NOT gate is an extension of the NOT gate in classical digital circuits.

A single input-output quantum gate can be represented by a 2×2 matrix. The state of a quantum state after passing through the quantum gate is determined by the value of the quantum state vector left-multiplied by the quantum gate matrix. The quantum gate matrix corresponding to the NOT gate is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Therefore, the result of a qubit passing a NOT gate is

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

2.2.2 **Hadamard Gate.** The Hadamard gate also works on a single qubit, which can decompose existing quantum states according to its coefficients:

$$H(\alpha|0\rangle + \beta|1\rangle) = \frac{\alpha + \beta}{\sqrt{2}}|0\rangle + \frac{\alpha - \beta}{\sqrt{2}}|1\rangle$$

Table 1: A Table-like Structure for Paper Organization

Introduction [Sec.1]		
Background [Sec.2]	Quantum bit (qubit) [Sec.2.1]	
	Quantum gate [Sec.2.2]	NOT gate [Sec.2.2.1]
		Hadamard gate [Sec.2.2.2]
		Controlled NOT gate [Sec.2.2.3]
	Quantum circuit [Sec.2.3]	
	Superposition and entanglement [Sec.2.4]	
	Quantum measurement [Sec.2.5]	
	Quantum algorithm [Sec.2.6]	
	Quantum computing [Sec.2.7]	
Quantum Software Engineering [Sec.3]	Quantum programming [Sec.3.1]	Concepts of quantum programming [Sec.3.1.1]
		Languages for quantum programming [Sec.3.1.2]
		Semantics of quantum programming [Sec.3.1.3]
	Definition of quantum software engineering [Sec.3.2]	
	Quantum software engineering methods, tools, and processes [Sec.3.3]	
	Generic view of quantum software engineering [Sec.3.4]	
	Quantum software life cycle [Sec.3.5]	
Quantum Requirements Analysis [Sec.4]		
Quantum Software Design [Sec.5]	Quantum software modeling [Sec.5.1]	
	Quantum software specification [Sec.5.2]	
	Pattern language for quantum software [Sec.5.3]	
	Quality attributes [Sec.5.4]	
Quantum Software Implementation [Sec.6]		
Quantum Software Testing [Sec.7]	Bug types in quantum software [Sec.7.1]	
	Assertions for quantum software [Sec.7.2]	Invariant and inductive assertion [Sec.7.2.1]
		Applied quantum Hoare logic [Sec.7.2.2]
		Assertion library for quantum software [Sec.7.2.3]
	Quantum software testing [Sec.7.3]	Open problems on testing quantum software [Sec.7.3.1]
		Fuzz testing [Sec.7.3.2]
		Property-based testing [Sec.7.3.3]
		Functional, white-box, and model-based testing [Sec.7.3.4]
	Quantum program debugging [Sec.7.4]	Debugging tactics [Sec.7.4.1]
		Assertion-based debugging [Sec.7.4.2]
		Debugging quantum processes [Sec.7.4.3]
		Language support for debugging [Sec.7.4.4]
	Quantum program analysis for bug detection [Sec.7.5]	
Quantum Software Maintenance [Sec.8]	Reengineering classical information system to quantum computing [Sec.8.1]	
Quantum Software Reuse [Sec.9]	Quantum pattern reuse [Sec.9.1]	
	Quantum circuit reuse [Sec.9.2]	
	Quantum state reuse [Sec.9.3]	
Challenge and Opportunities [Sec.10]	Quantum software requirements analysis [Sec.10.1]	
	Quantum software design [Sec.10.2]	Quantum architectural design [Sec.10.2.1]
		Detailed quantum design [Sec.10.2.2]
		Design models for quantum software [Sec.10.2.3]
	Quantum software implementation [Sec.10.3]	
	Quantum software reliability [Sec.10.4]	Fault model for quantum software [Sec.10.4.1]
		Quantum software testing [Sec.10.4.2]
		Quantum program debugging [Sec.10.4.3]
		Quantum software visualization [Sec.10.4.4]
		Quantum program verification [Sec.10.4.5]
	Quantum software maintenance [Sec.10.5]	
	Quantum software resue [Sec.10.6]	
Related work [Sec.11]	Quantum programming languages [Sec.11.1]	
	Quantum software engineering [Sec.11.2]	
	Quantum software development environments [Sec.11.3]	
Concluding Remarks [Sec.12]		

This can be represented by a matrix as:

$$H = \frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Although Hadamard gate is not directly related to the AND and OR gates in classical digital circuits, it has important applications in many quantum computing algorithms. Interested readers can try to prove that after applying the Hadamard gate twice in a row,
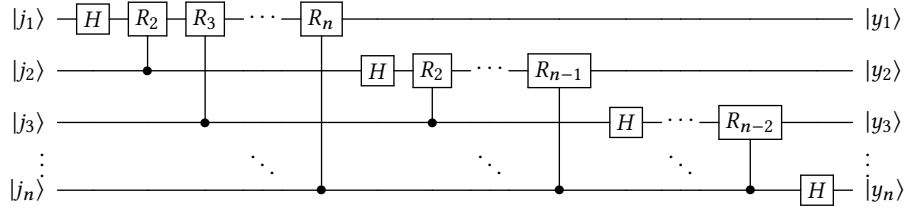
Figure 1: Quantum circuit for the quantum Fourier transform (QFT) algorithm.

the quantum state will return to its original state. This behavior is consistent with the NOT gate.

There can be an infinite variety of single input and output quantum gates, as long as the result of the left multiplication of the qubit state vector by the qubit state vector-matrix still satisfies the qubit normalization conditions.

2.2.3 **Controlled NOT Gate.** Computer programs are full of conditional judgment statements: if so, what to do, otherwise, do something else. In quantum computing, we also expect that the state of one qubit can be changed by another qubit, which requires a quantum gate with multiple inputs and outputs. The following is the controlled-NOT gate (CNOT gate). It has two inputs and two outputs. If the input and output are taken as a whole, this state can be expressed by

$$\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \theta|11\rangle$$

where $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ are column vectors of length 4, which can be generated by concatenating $|0\rangle$ and $|1\rangle$. This state also needs to satisfy the normalization conditions, that is

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\theta|^2 = 1$$

The CNOT gate is two-qubit operation, where the first qubit is usually referred to as the control qubit and the second qubit as the target qubit. When the control qubit is in state $|0\rangle$, it leaves the target qubit unchanged, and when the control qubit is in state $|1\rangle$, it leaves the control qubit unchanged and performs a Pauli-X gate on the target qubit. It can be expressed in mathematical formulas as follows:

$$CNOT(\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \theta|11\rangle) = \alpha|00\rangle + \beta|01\rangle + \gamma|11\rangle + \theta|10\rangle$$

The action of the CNOT gate can be represented by the following matrix:

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

2.3 Quantum Circuit

Quantum circuits, also known as quantum logic circuits, are the most commonly used general-purpose quantum computing models, which represent circuits that operate on qubits under an abstract concept. A quantum circuit is a collection of interconnected quantum gates. The actual structure of quantum circuits, the number and the type of gates, as well as the interconnection scheme, are all determined by the unitary transformation U, performed by the circuit. The result of a quantum circuit can be read out through quantum measurements.

As an example, Figure 1 shows the quantum circuit for the quantum Fourier transform (QFT) algorithm. The quantum gates used in the circuit are the Hadamard gate and the controlled phase gate $R_m = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^m}} \end{bmatrix}$. The black dots present the control bits.

2.4 Superposition and Entanglement

Quantum computers use the laws of quantum mechanics to provide a computation mechanism that is significantly different from classical machines. The first distinguishing feature of a quantum system is known as *superposition* [74, 308], or, more formally, the superposition principle of quantum mechanics. A quantum system is actually in all its possible states at a time, rather than existing in one distinct state at the same time. For a quantum computer, this means that a quantum register exists in the superposition of all its possible 0's and 1's configurations, which is different from a classical system whose register contains only one value at any given time. Until the system is observed, it collapses into an observable, deterministic classical state.

Quantum systems may also exhibit *entanglement* [79, 242], which is a quantum mechanical phenomenon. A state is considered to be entangled if it cannot be broken down into its more basic parts. In other words, two distinct elements of the system are entangled if one part of a system cannot be described without considering the other. A particularly interesting feature of quantum entanglement is that elements of a quantum system may be entangled even if they are separated by considerable space. Thus, measurements performed on one system can instantaneously influence other systems entangled with it. Quantum entanglement has applications in quantum computation [202, 257], quantum cryptography [26], and quantum teleportation [112, 139].

2.5 Quantum Measurement

From the introduction of the quantum gate above, we can see that a qubit is the superposition state of two quantum states $|0\rangle$ and $|1\rangle$; two qubits combined into a whole are in a superposition of four quantum states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. From this analogy, n qubits can be described by a superimposed state of 2^n qubits, which is a huge advantage over n classical bits that have only one fixed state. However, the laws of physics also have their limitations-in order to know the exact state of a qubit, a measurement is needed. But the measurement can cause the superposition state to collapse into a deterministic state. The information contained in a qubit after
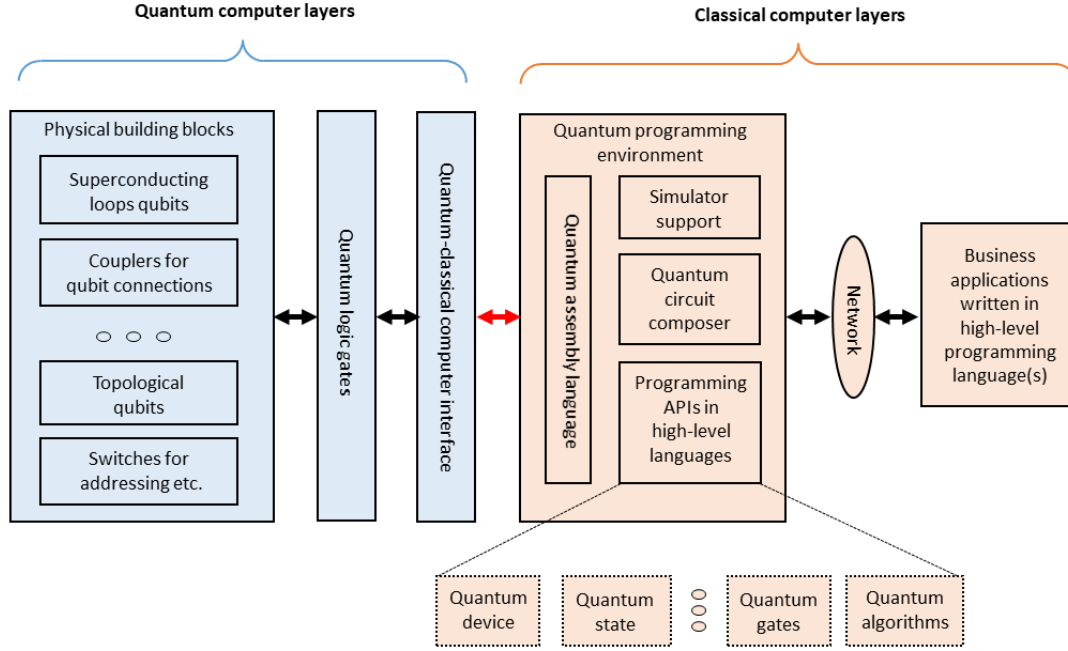
Figure 2: The architecture of quantum computing system in [260].

measurement is similar to a classical bit, and its value can only be 0 or 1. A superposed state can collapse to 0 or 1, the probability of which is determined by the coefficients α and β in the superposed state. The probability that this superposed state collapses to $|0\rangle$ or $|1\rangle$ is $|\alpha|^2$ and $|\beta|^2$, respectively. Similarly, for a two-qubit system, the probability that the measurement results in $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ is $|\alpha|^2, |\beta|^2, |\gamma|^2$, and $|\theta|^2$, respectively.

Such physical law causes the result of quantum computing to be non-deterministic. In actual applications, additional means are needed to verify the correctness of the output. For the algorithm used in quantum computing, if we can reduce the superposition state of qubits as much as possible in the previous step of measurement, we can get correct results with a high probability.

2.6 Quantum Algorithm

Quantum algorithms are designed to solve classical problems in a probabilistic manner [192]. For example, the phase estimation algorithm (QPE) [149] takes a matrix U, an eigenvector $|\psi\rangle$, as inputs, and calculates the eigenvector value of U corresponding to $|\psi\rangle$. Another example is Grover's search algorithm [118]: given a sparse non-zero function $f : \{0, \ldots, 2^{n-1}\} \rightarrow \{0, 1\}$, Grover's algorithm outputs a value x such that the probability of $f(x) = 1$ is high enough to beat brute force search, which is true in general. Quantum algorithms are usually designed to solve (classical) problems more efficiently than existing classical algorithms.

A quantum algorithm is generally realized on a quantum circuit designed according to the parameters of the problem (such as the size of the instance), usually through the following three steps of iteration:

(1) *Memory initialization*: preparing the initial state (basis state) $|x\rangle$ with $x \in \mathbb{B}^n$ (classical register),
(2) *Operation of the quantum circuit*: applying a quantum circuit of polynomially many in n gates from some universal gate set, and
(3) *Performing an elementary measurement*: measuring the quantum state to retrieve classic data.

Quantum circuits are regarded as a predictive tool that can provide some (classical) information probabilistically, from which the target result can be inferred. The fact that the probability is high enough is a direct consequence of the mathematical properties of the unitary mapping described by the quantum circuit. The essence of quantum algorithms (and the reason for their efficiency) is to describe an effective circuit that implements this unitary mapping. Therefore, it is crucial to describe an efficient circuit that can realize the unitary mapping in order to obtain the efficiency of the quantum algorithm.

2.7 Quantum Computing

The necessary procedure of quantum computing is as follows:

- Start with a set of calculated ground states (each qubit is initialized to $|0\rangle$ or $|1\rangle$ as input to the calculation);
- Pass the qubits through a series of quantum gates according to a predetermined algorithm;
- A series of bits is obtained as a result of quantum measurement.

Note that the realization of each quantum gate requires manual manipulation of quantum gates.

Table 2: A Brief and Historical Summary of Quantum Programming Languages

Year	Language	Reference(s)	Semantics	Host Language	Paradigm
1996	Quantum Lambda Calculi	[181]	Denotational	lambda Calculus	Functional
1998	QCL	[206–209]		C	Imperative
2000	qGCL	[241, 312–314]	Operational	Pascal	Imperative
2003	λ_q	[282, 283]	Operational	Lambda Calculus	Functional
2003	Q language	[32, 33]		C++	Imperative
2004	QFC (QPL)	[245–247]	Denotational	Flowchart syntax (Textual syntax)	Functional
2005	QPAlg	[141, 160]		Process calculus	Other
2005	QML	[10, 11, 113]	Denotational	Syntax similar to Haskell	Functional
2004	CQP	[102–104]	Operational	Process calculus	Other
2005	cQPL	[180]	Denotational		Functional
2006	LanQ	[188–191]	Operational	C	Imperative
2008	NDQJava	[298]		Java	Imperative
2009	Cove	[227]		C#	Imperative
2011	QuECT	[48]		Java	Circuit
2012	Scaffold	[1, 138]		C (C++)	Imperative
2013	QuaFL	[162]		Haskell	Functional
2013	Quipper	[114, 115]	Operational	Haskell	Functional
2013	Chisel-Q	[175]		Scala	Imperative, functional
2014	LIQUi|⟩	[292]	Denotational	F#	Functional
2015	Proto-Quipper	[234, 237]		Haskell	Functional
2016	QASM	[212]		Assembly language	Imperative
2016	FJQuantum	[82]		Feather-weight Java	Imperative
2016	ProjectQ	[122, 266, 272]		Python	Imperative, functional
2016	pyQuil (Quil)	[259]		Python	Imperative
2017	Forest	[61, 259]		Python	Declarative
2017	OpenQASM	[66]		Assembly language	Imperative
2017	qPCF	[213, 215]		Lambda calculus	Functional
2017	QWIRE	[217]		Coq proof assistant	Circuit
2017	cQASM	[146]		Assembly language	Imperative
2017	Qiskit	[4, 232]		Python	Imperative, functional
2018	IQu	[214]		Idealized Algol	Imperative
2018	Strawberry Fields	[147, 148]		Python	Imperative, functional
2018	Blackbird	[147, 148]		Python	Imperative, functional
2018	QuantumOptics.jl	[157]		Julia	Imperative
2018	Cirq	[271]		Python	Imperative, functional
2018	Q#	[269]		C#	Imperative
2018	$Q|SI\rangle$	[174]		.Net language	Imperative
2020	Silq	[35]		Python	Imperative, functional

* **Year**: The invented year of the language,
* **Language**: The name of the language,
* **Reference(s)**: The main reference paper(s) of the language,
* **Semantics**: The type(s) of semantics for the language the authors described in the reference(s),
* **Host language**: The classical language on (or to) which the language is based (or extended),
* **Paradigm**: We consider each language to belong to three types of paradigms: *imperative language*, *Functional language*, and *circuit design language*.

In [260], a general architecture of a quantum computing system has been proposed. As shown in Figure 2, the architecture comprises of two parts: *quantum layer* and *classical layer*. The quantum layer contains purely quantum hardware and circuitry, and can be considered as comprising the quantum processing unit (QPU). The detailed composition of this layer is listed as follows.

- *Physical building blocks* include quantum hardware that typically makes use of superconducting loops for the physical realization of qubits, and the physical qubit coupler/interconnect circuitry and other elements that are needed for qubit addressing and control operations.
- *Quantum logic gates* contain physical circuitry that makes up quantum logic gates.
- *Quantum-classical interface* includes the hardware and software which provides the interfacing between classical computers and a QPU.

The classical layer consists of classical hardware and software, as shown in the following:

- *Quantum programming environment* provides the quantum assembly language that is necessary for instructing a QPU, the programming abstractions for writing quantum programs in a high-level programming language, and the simulator support, as well as IDEs.
- *Business applications* include quantum software applications that are written to cater to business requirements.

3 QUANTUM SOFTWARE ENGINEERING

Quantum computing is not only a technological advancement, but can also be considered as a new general-purpose paradigm for software development, which can radically influence the way a software system is conceived and developed. This calls for new, quantum-specific, software engineering approaches. In this section, we first introduce quantum programming and define quantum software engineering. This will be followed by an introduction of a quantum software life cycle for quantum software development, which will be a baseline for discussing the state of the art of quantum software engineering activities in the rest of the paper.

3.1 Quantum Programming

Quantum programming is the process of designing and building executable quantum computer programs to achieve a particular computing result [187, 301]. Since the quantum programming efforts predate the other quantum software development techniques, we will focus first on quantum programming. Here, we briefly introduce the concepts, languages, and semantics of quantum programming. An excellent book written by Ying [301] covers similar material to this section, with more detailed discussions of the fundamentals of quantum programming.

3.1.1 **Concepts of Quantum programming.** A quantum program consists of blocks of code, each of which contains classical and quantum components. Quantum operations can be divided into *unitary* operations (reversible and preserve the norm of the operands), and *non-unitary* operations (not reversible and have probabilistic implementations). A quantum program executed on a quantum computer uses a quantum register of qubits to perform quantum operations, and a classical register of classic bits to record the measurements of the qubits' states and apply quantum operators conditionally [66]. Therefore, a typical quantum program usually consists of two types of instructions. One is called *classical instructions* that operate on the state of classical bits and apply conditional statements. Another is called *quantum instructions* that operate on the state of qubits and measure the qubit values.

3.1.2 **Languages for Quantum Programming.** Early quantum programming language development efforts focused on exploring the quantum Turing Machine (QTM) model proposed by Deutsch [70], but did not result in practical tools for programming quantum computers. This situation made the quantum circuit models quickly become the driving force for quantum programming. To build it as a practical language (rather than just designing circuits), Knill [152, 154] proposed a pseudocode notion for quantum programming and the model of a quantum random-access machine (QRAM) in which the quantum system is controlled by a classical computer. This model influenced the design of subsequent quantum programming languages. Ömer [206–209] developed the first practical quantum programming language QCL with a C-like syntax in 1998. Since then, many quantum programming languages have been designed and implemented in terms of different types of language paradigms for programming quantum computers, including qGCL [241, 312–314], LanQ [188, 191], Scaffold [1, 138], Q language [32, 33], NDQJava [298], Q# [269], $Q|SI\rangle$ [174], ProjectQ [122, 266], and Qiskit [4] for imperative quantum programming languages. Quantum lambda calculi [181], QFC (QPL) [244, 245, 247], QML [10, 11, 113], cQPL [180], [286], QuaFL [162], Quipper [114, 115], LIQUi|> [292], qPCF [213, 215], and IQu [214] for functional quantum programming languages, and QPAlg [141], qASM [212], QuECT [48], QWire [217], Quil [259] for other quantum programming languages paradigms. Table 2 gives a summary of quantum programming languages since it first emerged in 1996. In the table, we summarize each quantum programming language according to six categories: *year*, *language*, *reference*, *semantics*, *host language*, and *language paradigm*. For more information regarding quantum programming languages, one can also refer to survey papers discussed in Section 11.

3.1.3 **Semantics for Quantum Programming.** Semantics for quantum programming have also been studied extensively, and recently many approaches [43, 53, 56, 83, 101, 123, 180, 189, 190, 211, 245, 301, 302] have been proposed for describing the semantics of quantum programming languages. These approaches can be classified into three categories: *operational semantics*, *denotational semantics*, and *axiomatic semantics*. Although there is no survey for the semantics of quantum programming languages, readers can also refer to some survey papers [100, 140, 244, 302] on quantum programming languages for the detailed discussions of the quantum language semantics issues.

3.2 Definition of Quantum Software Engineering

While quantum programming languages are exciting developments, coding is not the primary source of problems in quantum software development. Requirements and design problems are much more common and costly to correct. Therefore, the focus on quantum software development techniques should not be limited to quantum programming issues, but should also focus on other aspects of quantum software engineering. The promise quantum software development methodologies hold for overcoming complexity during analysis and design and accomplishing analysis and design reuse is really significant. If it is accepted that quantum software development is more than quantum programming, then a whole new approach, including life cycle, must be adopted to address other aspects of quantum software development.

Software engineering is a problem-solving process with roots in behavioral, engineering, project management, computer science, programming, cost management, and mathematical science. According to [96], the software engineering process can be defined as follows:

"A software process can be defined as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that

are needed to conceive, develop, deploy, and maintain a software product."

For the definition of software engineering, the IEEE has developed a 'standard' definition [60] for classical software engineering as follows:

"(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)."

Fritz Bauer [201] first stated the definition of software engineering as following:

"The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

Although many definitions of software engineering [13, 38, 60, 88, 262, 297] have been developed since then, Bauer's definition is still the widely accepted one [226], and can serve as the basis for our definition of quantum software engineering in this paper.

In this paper, we define quantum software to include not only a set of executable quantum programs but also associated supporting libraries and documents needed to develop, operate, and maintain quantum software. By defining quantum software in this broader perspective, we hope to emphasize the necessity of considering timely documentation as an integral part of the quantum software development process. Inspired by the definitions of classical software engineering, we define quantum software engineering as following:

"Quantum software engineering is the use of sound engineering principles for the development, operation, and maintenance of quantum software and the associated document to obtain economically quantum software that is reliable and works efficiently on quantum computers."

In this definition, we would like to highlight three important issues in quantum software engineering. First, it is important to apply the "*sound engineering principles*" to quantum software development. Second, the quantum software should be built "*economically.*" Finally, the quantum software should be "*reliable*" and needs to work "*efficiently*" on quantum computers.

3.3 Quantum Software Engineering Methods, Tools, and Processes

Quantum software engineering, as its classical counterpart [75], can also be considered as a layered technology, which contains three elements: *methods*, *tools*, and *processes*.

Quantum software engineering *methods* provide the techniques for constructing the quantum software. They consist of a wide range of tasks, including the design of data structures, program architecture, algorithm procedure, coding, testing, and maintenance. Quantum software engineering *tools* provide automated or semi-automated support for these methods. Quantum software engineering *processes* are the foundation for the quantum software engineering. The process provides the glue that holds the methods and tools together and enables the rational and timely development of quantum software. They define the sequence in which methods

would be applied: the deliverables, the controls that help quality assurance and change coordination, as well as the milestones that enable quantum software managers to access the progress.

Different ways of combining these three elements of quantum software engineering would lead to different quantum software engineering models. The selection of a suitable model should be based on the nature of the project and the application, the methods and tools to be used, and the controls and deliverables that are required. Three typical examples discussed in [226] for classical software engineering are the classical life cycle (or waterfall model), the prototyping model, and the evolutionary model. Each of these models could be extended to the domain of quantum computing for supporting the development of quantum software systems. In Section 3.5, we will introduce a life cycle for quantum software, which is inspired by the idea of the waterfall model from classical software engineering.

3.4 A Generic View of Quantum Software Engineering

Pressman [226] introduced a generic view of classic software engineering, which can be extended to the field of quantum software engineering. Quantum software engineering can be regarded as a branch of systems engineering [50, 80], which involves the development of large and complex quantum software systems. To this end, the following issues must be discussed.

- What is the problem to be solved by quantum software?
- What characteristics of the quantum software are used to solve the problem?
- How will the quantum software (and the solution) be implemented?
- What method will be used to detect errors made in the design and construction of the quantum software?
- How will the quantum software be supported in a long period, when the users request corrections, adaptations, and enhancements?

To adequately engineering quantum software, a quantum software engineering process must be defined. In this section, we consider the generic characteristics of the quantum software process. The work related to quantum software engineering can be categorized into three generic phases: *definition*, *development*, and *support* phases. These phases are independent of the application domain, project size, or complexity.

The definition phase focuses on *what*, dealing with the problems that quantum software developers try to determine. These issues include topics such as what information to process, what functions and performance are required, what interfaces to build, what design constraints exist, and what verification standards are needed to define a successful quantum software system. At this stage, there may be three sub-processes, including quantum software system analysis, quantum software project planning, and quantum software requirements analysis.

The development phase focuses on *how*, which deals with the problems that quantum software developers try to describe. These issues include how to design quantum software architecture and related data structures, how to implement quantum program details, how to translate quantum software design into a quantum
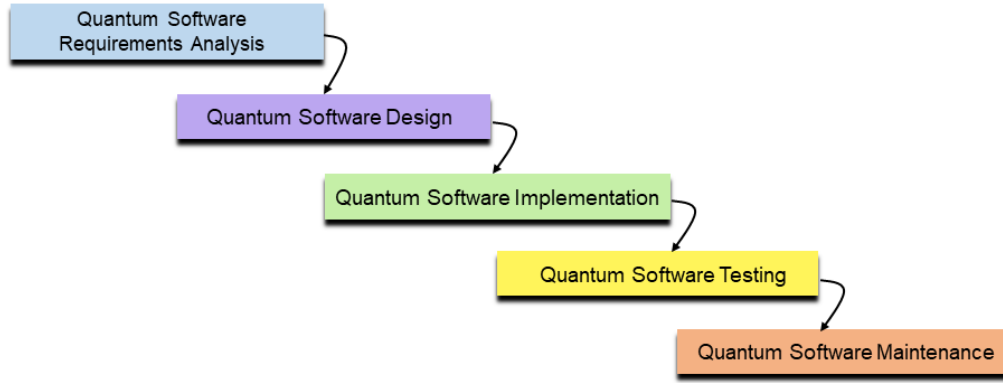
Figure 3: A quantum software life cycle.

programming language, and how to conduct quantum software testing. This phase may also contain some sub-processes, including quantum software design, quantum software implementation, and quantum software testing.

The maintenance phase focuses on *change*, which is related to error correction, adaptions required as the quantum software environment evolves, and modifications due to enhancements brought about by changes in customer requirements. The maintenance phase reapplies the definition and development phases, but it is carried out in the context of existing quantum software.

3.5 Quantum Software Life Cycle

A software life cycle model can be defined as a reference model for a software development process that supports the design and construction of high-quality software [105]. A software life cycle provides a well-structured flow of phases that can help an organization to quickly produce high-quality software that is well-tested and ready for production use. Several software life cycle models have been proposed for classical software development, including the waterfall model [238], evolutionary model [128], and the Spiral model [39]. Among them, the most widely accepted life cycle model for classical software development is the waterfall life cycle model, which is sometimes called the classical model. Other models are often improved upon it.

As the first step, this paper introduces a systemic, sequential life cycle model for the quantum software development process to support the design and construction of quantum software systems. A quantum software life cycle model begins at the system level and progresses through requirements analysis, design, implementation, testing, and maintenance. The model, however, is extensible in the sense that one can add more phases into it if necessary. Figure 3 shows the life cycle, which encompasses the following phases, each one is flowing into the next.

- *Quantum software requirements analysis*
- *Quantum software design*
- *Quantum software implementation*
- *Quantum software testing*
- *Quantum software maintenance*

In the following, we briefly introduce each phase of the life cycle model from the perspective of quantum software development.

3.5.1 **Quantum Software Requirements Analysis.** The quantum software life cycle model begins with the requirements analysis phase, where the stakeholders discuss the requirements of the software that needs to be developed to achieve a goal. The requirements analysis phase aims to capture the detail of each requirement and to make sure everyone understands the scope of the work and how each requirement is going to be fulfilled. The analysis creates a set of measurable quantum software requirements that specify, from the supplier's perspective, what characteristics, attributes, and functional and performance requirements the quantum software system is to possess, to satisfy stakeholder requirements. Later life cycle phases, including design, implementation, testing, maintenance for quantum software, assume that requirements analysis continues through the quantum software life cycle.

3.5.2 **Quantum Software Design.** Quantum software design is the second phase in the life cycle, which is a process to transform user requirements into a suitable form that helps the programmer in quantum software implementation (coding).

As classical software design [75], quantum software design may also involve two stages: *architectural design* and *detailed design*. *Architectural design* defines a collection of quantum software components, their functions, and their interactions (interfaces) to establish a framework for the development of a quantum software system. *Detailed design* refines and expends the architectural design to describe the internals of the quantum software components (the algorithms, processing logic, data structures, and data definitions). Detailed design is complete when the description is sufficient for implementation (coding).

3.5.3 **Quantum Software Implementation.** After completing the requirements and design activities, the next phase of the life cycle is the implementation or development of quantum software. At this phase, developers start coding based on the requirements and design discussed in the previous phases. Developers also perform the unit testing for each component to test the new code they write, review the code for each other, create builds, and deploy

Table 3: Brief Summary of the Patterns for Quantum Algorithms in [166]

Pattern Type	Description
(1) Initialization (aka state preparation)	Initializing the input of a quantum register in a straightforward manner
(2) Uniform superposition	Creating an equally weighted superposition of all possible states of the qubits of a quantum register
(3) Creating entanglement	Creating an entangled state
(4) Function table	Computing a function table of a finite Boolean function
(5) Oracle (aka black box)	Reusing the computation of another quantum algorithm
(6) Uncompute (aka unentangling aka copy-uncompute)	Removing entanglement that resulted from a computation
(7) Phase shift	Distinguishing important aspects of a state efficiently
(8) Amplitude amplification	Increasing the probability of finding a solution
(9) Speedup via verifying	Achieving a speedup when verifying a solution is simple
(10) Quantum-classic split	Splitting a solution between a quantum computer and a classical computer

quantum software to the environment. This development cycle is repeated until the requirements are met.

3.5.4 **Quantum Software Testing.** Testing is the last phase of the quantum software life cycle before the software is delivered to customers. During testing, testers start to test the quantum software according to the requirements. The aim is to find defects within the software as well as to verify whether the software behaves as expected based on the documentation from the quantum software requirements analysis phase.

3.5.5 **Quantum Software Maintenance.** Quantum software maintenance is the last phase of the quantum software life cycle, which represents all the modifications and updates made after the delivery of quantum software products.

4 QUANTUM SOFTWARE REQUIREMENTS ANALYSIS

To the best of our knowledge, until recently, no research work has been proposed to address the issues on quantum software requirements analysis. However, as quantum software development resources are going to be accumulated, we believe that the development of methodologies, techniques, and tools to support quantum software requirements analysis will become a critical and inevitable issue.

5 QUANTUM SOFTWARE DESIGN

Developing quantum algorithms is considered extremely difficult comparing to classical counterparts due to some intrinsic features of quantum computing, such as superposition and entanglement. Hence, new design principles for quantum algorithms are strongly demanded. Although software design is recognized as a critical phase in classical software engineering, researches [46, 220, 274] on the principle and methodology for quantum software design are just starting. This section gives an overview of state of the art on quantum software design.

5.1 Quantum Software Modelling

The Unified Modeling Language (UML) [40, 135, 240] is a general-purpose, well-known modeling language in the field of classical software engineering. It provides a standard way to visualize the

design of the classical software development life cycle. It seems reasonable to extend the UML approach to support quantum system design. Recently, Pérez-Delgado and Perez-Gonzalez [220] presented an approach to extending the UML to model quantum software systems. The extension covers two types of UML diagrams: *class diagram* and *sequence diagram*. Extension for the other three components (i.e., composite structure diagrams, activity diagrams, and state machine diagrams) still needs to be studied. They also shared three fundamental observations from their work, including:

- The nature of quantum computation, leading to the intrinsic difference between quantum and classical computation is in how it achieves its goal,
- Quantum computation changes the very nature of information itself, and is much more productive and powerful than classical computation, and
- The classical vs. quantum nature of the information used by a module is an important consideration when discussing about its internal implementation and interface.

Their observations, though not complete, can serve as a base for further studying the design methodologies of quantum software systems.

5.2 Quantum Software Specification

Quantum computing relies on quantum mechanics, which is a subject more familiar to physicists rather than computer scientists and software engineers. Thus, we must be aware of the underlying theory before we reason about quantum computers. Even though delivering the principles of quantum mechanics is not an easy task, due to their counter-intuitive nature.

Recently, the method for reasoning about quantum computing is a mixture of linear algebra and Dirac representation, which is a subject more suitable for physicists than computer scientists (software engineers) [46]. Therefore, it is necessary to provide a more "intuitive" way to think and write quantum algorithms, thereby simplifying the design and implementation of quantum software. This can be achieved, for example, by introducing a specification language, which adopts the symbolism and reasoning principle of software engineering and will take on the role of Hilbert space algebra, allowing us to describe quantum structures and design quantum algorithms in a more natural way [113].

Table 4: The Effects of QCS characteristics on Quality Attributes [260]

QCS Characteristics	Quality Attributes	Availability	Interoperability	Maintainability	Manageability	Performance	Reliability	Scalability	Security	Testability	Usability
Platform heterogeneity		U	–	U	U	–	U	–	–	U	–
Special physical environment		U	–	–	U	–	U	U	U	U	–
Large form factor		–	–	–	–	–	–	U	–	–	–
Higher energy efficiency		–	–	–	–	F	–	F	–	–	–
Lower level of the programming abstractions		U	–	U	–	–	U	–	–	U	–
Remote software development and deployment		–	–	U	–	–	–	–	–	U	–
Dependency on quantum algorithms		–	U	U	–	F	–	F	–	U	–
Limited portability of software		U	U	U	–	–	–	U	–	–	–
Limited quantum networking		U	–	–	–	U	U	U	–	–	–
Lack of native quantum operating system		–	–	–	U	U	U	U	U	–	–
Fundamentally different programming model		–	U	U	–	–	U	–	U	U	–
Dependency on classical storage		–	–	–	U	U	U	U	–	–	–

* Cell value indicates an impact on quality attributes: *"F": Favorable, "U": Unfavorable, "−": Unknown/Neutral.*

To this end, Cartiere [45, 46] presented some work on defining a formal specification language for quantum algorithms. The goal is to introduce a formal specification language that can be used to represent the basic notations of quantum computing, which are more suitable for quantum computing scientists and quantum software engineers. The language is based on Z [2, 296], a well-studied formal specification language for classical software systems. The language can represent some elementary quantum gates such as *Identity gate, Pauli-X gate, Phase Shift gate, Pauli-Z gate, Hadamard gate*, and *C-NOT gate*, which perform unitary transformations to support writing quantum programs. Cartiere also showed how to use the language to specify some simple quantum algorithms, such as the Deutsch-Jozsa algorithm [70, 71]. It is, however, unclear if the language can be used to represent more complex quantum algorithms such as Shor's *integer factoring* quantum algorithm [257] and Grover's *database search* quantum algorithm [118].

5.3 Pattern Language for Quantum Software

In classical computing, software engineers have long noticed that specific patterns recur and persist across different applications and systems. A pattern is a systematic design that can capture the experiences of experts on good designs or best practices, and record the essence of this wisdom in an easy-to-understand way for peers to use [12]. Patterns can be used to describe and reason about the relationship between a specific context, and a problem and its solutions in that context. A pattern language is a network of related patterns that may jointly contribute to an encompassing solution of a complex problem [5, 23]. Patterns and pattern languages are used in multiple design and engineering disciplines, especially for software architecture design [44, 93, 95, 287].

Leymann [166] proposed a pattern language for supporting the development of quantum algorithms. There is a need for documenting solutions for problems recurred in quantum software development, as observed and mentioned in [171, 202], which informally summarized some basic "tricks" used in quantum algorithms. The main contribution of this work is to systematize this to become a subject of a software engineering discipline for quantum algorithms. In [166], Leymann identified ten types of basic patterns derived from existing quantum algorithms that mainly based on gate models. Each pattern is described through eight elements, including *name, intend, icon, problem statement, context, solution, know uses,* and *next*. The summary of these ten types of patterns is listed in Table 3.

Additionally, Leymann also discussed some issues on the using of these patterns regarding software engineering, and mentioned that the pattern language proposed can be stored as a pattern repository. In the repository, each pattern could be linked with the corresponding implementation in a concrete quantum programming language to support the programming of the pattern. Note that the patterns identified in [166] is in no way meant to be exhaustive, and more patterns should be identified in the future to make the patterns and the pattern language practical useful.

5.4 Quality Attributes

With practical quantum computing systems (QCSs) becoming a reality rapidly, it is desirable to make use of their real potential in software applications. Therefore, it is crucial to determine the implications of QCSs for quantum software architecture, and some questions must be answered, like "*What are the key characteristics of QCSs, which are efficacious for quantum software development?*" and "*In what way does a QCS affect the quality attributes (and non-functional requirements) of quantum software applications?*"

To answer these questions, Sodhi [260] presented an in-depth study on the state-of-the-art QCSs for identifying all such characteristics of a QCS that are relevant from a software architecture perspective, and performed an investigation on how these characteristics may affect the architecture of a quantum software application. As the first step, Sodhi sought and investigated related

papers and software artifacts of QCSs and identified thirteen critical characteristics of QCSs. This includes platform heterogeneity, physical environment, large form factor, energy efficiency, lower level of the programming abstractions, remote software development and deployment, dependency on quantum algorithms, limited portability of software, limited quantum networking, lack of native quantum operating system, limited multitasking and multiprocessing, fundamentally different programming model, and dependency on classical storage. These key characteristics of QCSs form a base for further studies on how they could affect the quality attributes of the software architecture of a QCS.

Quality attributes (QAs) are measurable or testable properties of a system that is used to indicate how well the system satisfies the needs of its stakeholders [21]. They, therefore, have a system-wide impact on the architecture, implementation as well as the operation of the system. QAs are usually categorized in various specialized areas, such as design qualities, runtime qualities, system qualities, user qualities, non-runtime qualities, architecture qualities, and business qualities [41, 216]. It is recognized that when software applications have at least a reasonable level of quality attributes such as performance, reliability, scalability, security, and usability, the overall design and quality of the applications can be considered good [21, 22].

For investigating the impact of QCS characteristics on the QAs, Sodhi [260] adapted a slightly expanded list of QAs, which include availability, interoperability, maintainability, manageability, performance, reliability, scalability, security, testability, and usability. He only discussed the parts of a QA that are related to determine how this QA is affected by the characteristics of QCSs. To do so, one may take each characteristic of QSC and considers how it affects the various QAs. The impact of each characteristic on QAs is classified as *favorable*, *unfavorable*, or *neutral*. The summary of the impact of various characteristics of QCSs on QAs is shown in Table 4, and the detailed discussions on those impacts can be referred to the paper [260]. Additionally, several issues should be further studied, such as identifying more characteristics of QCSs to enhance the current list and investigating more QAs for obtaining more complete findings (insights) for the problem studied in the paper. In conclusion, Sodhi [260] argued that the evolution of the technology would likely introduce additional concerns and factors that may affect the architecture of quantum software application, as quantum computing is undergoing rapid development.

6   QUANTUM SOFTWARE IMPLEMENTATION

This section will be brief, as much of the material has been covered in the survey (or overview) papers of quantum programming languages [54, 98, 102, 186, 244, 261, 263, 280, 281, 301, 302, 311].

The initial focus of quantum software development on the implementation level has resulted in a range of quantum programming approaches. It is crucial to identify a suitable quantum programming technique when implementing a quantum application. Most quantum programming techniques support one or more base programming languages. For example, languages are available for quantum programming in C (QCL), C++ (Scafflod), C# (Q#), Python (ProjectQ, Qiskit, Forest), F# (LIQUi|⟩), Scala (Chisel-Q), and Haskell

(Quiper). Therefore, the first natural step in the choice of a suitable technique is to reduce the available set of techniques to those that support the base programming language to be employed in application development. If no suitable technique is available for the base programming language, a custom language needs to be implemented.

7   QUANTUM SOFTWARE TESTING

Quantum computers are powerful. However, since the human intuition is much better adapted to the classical world than the quantum world, programmers might make mistakes more frequently when designing programs on quantum computers compared to programming on classical counterparts [300]. Furthermore, compared to the classical computers, quantum computers have very different properties, such as quantum superposition, entanglement, and no-cloning [202]. Therefore, the prediction of the behavior of quantum software is difficult [54, 300]. As a result, new quantum software testing and debugging techniques are needed to discover [268]. Recently, researches are emerging for identifying bugs, and for testing and debugging of quantum software. This section gives an overview of state of the art on quantum software testing, in a broader sense, according to quantum software *bug type*, *assertion*, *testing*, *debugging*, and *analysis*.

7.1   Bug Types in Quantum Software

A software bug is regarded as the abnormal program behaviors which deviate from its specification [6]. Fault models are a common concept in the software and hardware testing field. To debugging and testing quantum software, it is essential to define some fault model which can help deeply understanding the behavior of bugs in quantum programs. Several researches considered this issue recently. To support quantum software debugging, Huang and Martonosi [132, 133] studied the bug types for special quantum programs. Based on the experiences of implementing some quantum algorithms, they identified several types of bugs specific to quantum computing. These bug types include incorrect quantum initial values, incorrect operations and transformations, incorrect composition of operations using iteration, incorrect composition of operations using recursion, incorrect composition of operations using mirroring, incorrect classical input parameters, and incorrect deallocation of qubits. They also proposed some defense strategies for each type of bugs. The detailed summary of these bugs and their corresponding strategies can be found in Table 5.

7.2   Assertions for Quantum Software

An assertion is a statement about the expected behavior of a software component that must be verified during execution [92]. In software development, a programmer defines an assertion to ensure a specific program state at run time. Assertions have been used extensively for detecting runtime faults, documenting programmer intent, and formally reasoning about the correctness of classical programs [58]. Recently, several researches have been carried out for defining and identifying assertions in quantum software systems.

7.2.1   **Invariant and Inductive Assertions.** Ying *et al.* [303] studied the issue of how to define the notion of invariant and inductive assertion [91] for quantum programs. They considered this

Table 5: Bug types and their corresponding defense types for quantum software in [132, 133]

Bug Type	Defense Strategy
(1) Incorrect quantum initial values	Assertion checks for classical and superposition preconditions
(2) Incorrect operations and transformations	Assertion checks for unit testing
(3) Incorrect composition of operations using iteration	Assertion checks for classical intermediate states
(4) Incorrect composition of operations using recursion	Assertion checks for entangled intermediate states
(5) Incorrect composition of operations using mirroring	Assertion checks for product state postconditions
(6) Incorrect classical input parameters	Assertion checks for classical postconditions
(7) Incorrect deallocation of qubits	Assertions on algorithm postconditions

issue in two different ways – from the perspectives of additive and multiplication. They proved that both additive and multiplication invariants could be used to prove the partial correctness of quantum programs. They also showed that additive invariants could be derived from additively inductive assertions, and can be generated through an SDP (Semidefinite Programming [284]) solver. However, how to generate the multiplication invariants is still an open problem that has not been addressed in [303], and therefore still needs to be explored.

7.2.2 **Applied Quantum Hoare Logic.** Hoare logic (also known as Floyd-Hoare logic) [90, 129] is a formalism with a set of logic rules used for rigorous reasoning on the correctness of computer programs. Hoare logic has been extensively used for the verification, debugging, and testing of classical software [3, 14]. Recently, researchers have extended Hoare logic to the quantum domain to support the formal verification of quantum programs [18, 43, 73, 143, 277, 300, 310]. Among them, Li *et al.* [310] introduced applied quantum Hoare logic (aQHL), which is a simplified version of quantum Hoare logic (QHL) [300], with particular emphasis on supporting debugging and testing of the quantum programs. aQHL simplified two issues of QHL through: (1) binding QHL to a particular class of pre- and post-conditions (assertions), that is, projections, to reduce the complexity of quantum program verification and to provide a convenient way used for debugging and testing of quantum software. (2) providing additional rules for reasoning about the robustness of quantum programs. As aQHL can be used to specify assertions such as pre- and post-conditions, and invariants for quantum software, hopefully, it could provide a very efficient way to support assertion-based quantum software debugging and testing. Although no detail [310] was given on how aQHL could be used to debug and test quantum software, we believe it could be a promising way for tracking errors in quantum software, similar to its classical counterpart. Moreover, besides aQHL, similar approaches that use other types of (dynamic) quantum logic [18, 43, 143] for specifying assertions for supporting debugging of quantum software should also be investigated.

7.2.3 **Assertion Library for Quantum Software.** As we will discuss in Section 7.3, Honarvar and Nagarajan [131] proposed a property-based approach to testing quantum programs in Q# (developed by Microsoft) [269]. To this end, they developed a library of assertion methods that can be used to specify various properties (assertions) in Q# programs for testing. These assertion methods include `AssertProbability`, `AssertEntangled`, `AssertEqual`, `AssertTeleported` and `AssertTransformed`. `AssertProbability` uses a robust statistical method to test the probability of observing a qubit in a given state. `AssertEntangled` takes two qubits as its arguments to test whether or not they are entangled. `AssertEqual` tests the equality of two states in quantum programs. `AssertTeleported` tests quantum teleportation as it is a significant protocol in the quantum realm. `AssertTransformed` tests the validity of any unitary transformation. The detailed description of each assertion method is summarized in Table 6. Although those assertion methods are used initially for testing, they also have the potential to be used in quantum program debugging and runtime error checking.

7.3 Quantum Software Testing

Testing [24, 196] is the process that executes a program with the intent to find errors, which is a very critical process for supporting quality assurance during software development. In this section, we review state of the art for testing quantum software.

7.3.1 **Open Problems on Testing Quantum Software.** Miranskyy and Zhang [184] proposed some challenges associated with white-box and black-box testings as well as verification and validation of quantum programs. They have shown that some of the existing software engineering practices are readily transferable to the quantum computing domain (e.g., code review). Others are difficult to transfer (e.g., interactive debugging). And the rest have to be introduced (e.g., complexity-dependent placement of a validation program). Rather than proposing a particular testing method (or strategy) for quantum software, they tried to define the software engineering practices for quantum software. With this definition at hand, software engineering specialists in academia and industry can start exploring this fascinating area of quantum computing and expand their work to other areas of testing as well as the remaining phases of the quantum software life cycle.

7.3.2 **Fuzz Testing.** Fuzz testing (or fuzzing) [270] is a software testing technique that inputs invalid or random data called *fuzz* into the software system to discover coding errors and security loopholes. Wang *et al.* [289] adapted an existing technique from classical software testing called coverage-guided fuzzing (CGF) [65, 248, 306] to test quantum software. They proposed `QuanFuzz`, a search-based test input generator for quantum software. The basic idea of `QuanFuzz` is to define some quantum sensitive information to evaluate the test inputs for quantum software and use a matrix generator to generate test cases with higher coverage. The testing process of `QuanFuzz` consists of two steps:

Table 6: Brief Summary and Comparison of Assertion Methods in [131] and Assertion Functions in [172]

Assertion Methods [131]	Description
`AssertProbability(q0,state,probability)`	To take three arguments to test the expected `probability` of observing a qubit `q0` in a given `state` after measurement.
`AssertEntangled(q0,q1)`	To take two qubits `q0` and `q1` as its arguments to test whether or not they are entangled.
`AssertEqual(q0,q1)`	To take two qubits `q0` and `q1` as its arguments to compare the equality of these two states.
`AssertTeleported(q0,q1)`	To take the sent and received qubits `q0`, `q1` as its arguments to test quantum teleportation.
`AssertTransformed(q0,(θ interval)(φ interval))`	To test the validity of any unitary transformation.
Assertion Functions [172]	**Description**
`classical_assertion(circuit,qubitList,value)`	To take three arguments to specify the quantum circuit under test, the list of qubits for assertion, and a particular classical value to assert for.
`entanglement_assertion(circuit,qubitList,flag)`	To take three arguments specifying the quantum circuit under test, the list of qubits for assertion, and the type of entanglement.
`superposition_assertion(circuit,qubitList,phaseDict,flag)`	To take four arguments specifying the quantum circuit under test, the list of qubits for assertion, the quantum state dictionary for the qubits, and a flag.

(1) Extracts quantum sensitive information from the quantum source code. Such information includes the measurement operations on the quantum registers and the sensitive branches associated with the measurement results.

(2) Uses the sensitive information guided algorithm to mutate the initial input matrix and selects those matrices which improve the probability weight for a value of the quantum register to trigger the sensitive branch.

This process keeps iterating until the sensitive branch is triggered. They also implemented `QuanFuzz` based on $Q|SI\rangle$ [174] and *Nrefactory* for quantum software simulation and code instrumentation respectively, and evaluated `QuanFuzz` on seven quantum programs with different registers (containing 2 to 8 qubits), which are build-in benchmark programs from $Q|SI\rangle$. The experimental result showed that `QuanFuzz` can obtain 20%~60% more branch coverage than the classical test input generation method.

7.3.3 **Property-Based Testing.** Property-based testing [55, 87] uses specifications of essential properties to produce testing criteria and procedures which focus on these properties in a systematic manner. Honarvar and Nagarajan [131] presented QSharpCheck, a property-based testing approach for quantum software written in Q#. To this end, they defined a testing specification language to specify the properties of Q# programs. The whole design of the language is inspired by several ideas, such as the syntax of Q#, quantum predicates and predicate transformers by D'Hondt and Panangaden [73], and quantum Hoare logic by Ying [300]. In the language, a test is represented by four parts: test property name and parameters, allocation and setup, function call, and asset and deallocate. They also identified several types of assertions that formed the basis of post-conditions and assertion types associated with the test specification language. Based on these specified properties, one can generate various types of test cases, run the test cases to test Q# programs, and check the output results to see if there are any problems within the programs. Moreover, they also showed some basic considerations on the design and implementation of QSharpCheck tool and carried out two case studies to demonstrate the effectiveness of QSharpCheck through mutation analysis of Q# programs.

7.3.4 **Functional, White-box, and Model-based Testing.** Usaola [278] proposed some ideas and identified some challenges when applying classical software testing to the quantum computing domain. He discussed how some prevalent strategies of classical software testing, such as *functional testing*, *white-box testing*, and *model-based testing*, can be applied to test quantum software. Functional testing [197] is a type of software testing that validates the software system against the functional requirements/specifications. Usaola discussed some basic concepts regarding functional testing (such as test case and test suite) and showed that a process of functional testing for quantum software might consist of the following three steps:

(1) The initial situation of a quantum test case sets up the initial status of the qubits.

(2) Similar to classical testing, the quantum circuit is executed.

(3) The test suite saves the obtained result to calculate the most probable result.

White-boxing testing [197] is a method that tests a software solution's internal structure, design, and coding. Among different white-box testing methods, Usaola specially studied mutation testing, which might be a good candidate technique for quantum software testing. He also gave a simple example program of adding two integer numbers using IBM's Qiskit simulator to show how the mutation testing can be applied for testing. However, how to define suitable quantum mutant operators to support quantum mutation testing has not been discussed. Model-based testing [279] is a software testing technique where run time behavior of the software during the test is checked against predictions made by a model. To apply model-based testing to the quantum software, one should first use some modeling language, for example, UML, to model the behavior of the quantum software. Then, based on the model, one can generate test cases for the software under test to perform the testing. As we discussed in Section 5, one may use the Q-UML [220], an extension of UML to the quantum domain, to model a quantum software system to support the test case generation of the system.

Q# [269] is a quantum programming environment developed by Microsoft that supports quantum programming and quantum software development. Q# offers tools for testing and debugging quantum programs [273]. The tools can be used to verify that quantum programs act as intended as well as to diagnose an incorrect quantum program.

Although testing is a widely used technique to guarantee the reliability of a software system, as Dijkstra stated in [68], *testing*

can be used to show the presence of bugs, but never to show their absence. Therefore, after testing, systematical techniques such as *debugging* is still needed to localize the bugs in the system.

7.4 Quantum Program Debugging

The process of finding and fixing bugs is called debugging, which is the activity of analyzing the program to locate and correct errors in a program by reasoning about causal relations between bugs and the error detected in the program [28, 42]. Program debugging tools are essential for any programming environment [15, 165]. In the life cycle of software development, almost 25% of maintenance is carried out for debugging [170]. Debugging methodologies and techniques are also crucial for quantum programming [54]. This section gives an overview of quantum program debugging techniques from four aspects: *debugging tactics, debugging quantum processes, assertion-based debugging*, and *language support for debugging.*

7.4.1 **Debugging Tactics.** For classical program debugging, some well-established common tactics including *backtracking, cause elimination*, and *brute force* [197, 226] are widely used. These debugging tactics can also be used to support the quantum program debugging.

Miranskyy *et al.* [185] analyzed the debugging tactics for quantum programs and discussed the possibility of applying some well-established classical debugging techniques to the quantum domain. They mainly considered three types of debugging tactics in classical debugging, as mentioned above, and concluded that *backtracking* tactics, especially those based on code reviews and inspections, could probably be applied for quantum debugging. This is also confirmed by the discussions with practical quantum software developers who usually use code reviews and inspections for their daily programming tasks [185].

For *cause elimination* tactics, they pointed out that it could be naturally extended to the domain of quantum computing. First, one can set a hypothesis and specify a root cause for a bug understudy during quantum debugging, and then collect data and perform some experiments based on the data to prove the hypothesis. However, considering the probabilistic nature of the quantum program's behavior [184, 202], the final result of the quantum program should be assessed based on the distribution of the results obtained from multiple executions of the program. To solve the problem, Miranskyy *et al.* [185] suggested to apply classical testing techniques of probabilistic programs [77, 78] to the quantum computing domain.

For *brute force* tactics, the most common tactic for classical debugging, its applicability to quantum debugging might depend on two situations. If we regard the quantum program as a black box, one could follow the classical debugging process: (1) First, trace the input and the output of the program (2) Then, record the input and output data in a log file. (3) Finally, analyze and compare these data against expected values to see if they are consistent. If we treat the quantum program as a white box, one has to consider how to capture the execution traces during the execution of the program to perform interactive debugging. However, considering the specific features such as superposition, entanglement, and no-cloning, in quantum computing, it is almost impossible. Miranskyy *et al.* [185] discussed some scenarios regarding those specific quantum features for which the classical debugging cannot be applied, and suggested

some potential solutions to solve these problems. Those solutions, however, might still be immature, thus may be infeasible in practice. Therefore, much work remains to be done to work out some solutions to support quantum program debugging.

7.4.2 **Assertion-based Debugging.** Several assertion-based approaches have been proposed recently for debugging quantum programs. As we described previously, Huang and martonosi [132, 133] summarized several types of bugs in quantum programs based on their experiences of implementation of a set of quantum programs (or algorithms [118, 182, 204, 257]). They also proposed some corresponding defense strategies in terms of programming and assertion to prevent such bugs during programming to develop bug-free quantum programs. This study is preliminary, and further, Huang and Martonosi [133] presented the statistical assertions for quantum programs based on statistical tests on some classical observations. These allow the programmers to decide if a quantum program state matches its expected value in one of classical, superposition, or entangled types of states. Based on this, they classified possible assertions in quantum software into three types: *classical assertion, superposition assertion*, and *entanglement assertion*. They extended an existing quantum programming language called Scaffold [1, 138] with the ability to specify quantum assertions, and developed a tool to check these assertions in a quantum program simulator called QX [145]. To demonstrate the effectiveness of their assertion-based approach, they performed three case studies on three benchmark quantum programs: factoring integers, database search, and quantum chemistry, each of which represents a different class of quantum algorithms. They also showed some results on what types of bugs are possible and laid out a strategy for using quantum programming patterns to place assertions and prevent bugs. Moreover, to validate the proposed approach, they cross-validated the quantum programs and the simulation results through the functional equivalent programs implemented in different quantum programming languages, including LIQUi|⟩ [236], ProjectQ [122, 266], and Q#[269]. Through this validation, they found and shared some results and insights on how language features of different quantum programming languages can help insert the quantum assertions in a suitable place in the program, or otherwise prevent bugs in the first place.

Zhou and Byrd [309] and Liu *et al.* [172] observed that the critical limitation of Huang and Martonosi's assertion-based approach [133] is that each measurement during debugging has to stop the execution of the program, and the assertions require aggregates of runs when the actual computation results are to be measured. To overcome this limitation, motivated by quantum error correction (QEC) [110, 202] and nondestructive discrimination (NDD) [136], they proposed an approach to constructing suitable quantum circuits to support runtime assertion check of quantum software [172, 309]. The key idea of this approach is to use ancilla qubits (some additional quantum bits) to collect the information of the qubits under test indirectly and to measure those ancilla qubits, rather than the original qubits under test directly. This can avoid disrupting the program execution during the assertion check at runtime. To this end, they identified three types of dynamic assertions, respectively, for classical values, entanglement, and superposition. *Assertion for classical value* is to ensure that the qubits are initialized

to the correct value, or some intermediate classical results should satisfy some condition such as ($|\psi\rangle$==$|0\rangle$)). *Assertion for entanglement* is to assert two or more qubits are in entangled states based on checking the parity of two qubits. *Assertion for superposition* is to assert that the use of Hadamard gates to set the input qubits in the uniform superposition state, and also to assert arbitrary superposition state represented as $|\psi\rangle = \sin(\frac{\theta}{2})|0\rangle + e^{i\varphi}cos(\frac{\theta}{2})|1\rangle$.

The implementation of the idea of assertion circuits is also discussed. This is based on Qiskit developed by IBM [232], an open-source framework for quantum computing, through integrating three types of assertion functions into the Qiskit development environment. Programmers can use these functions to instrument the necessary assertion circuits for classical, entanglement, and superposition states, and therefore can check the corresponding ancilla qubits for error detection. These three kinds of assertion functions include `classical_assertion`, `entanglement_assertion`, and `superposition_assertion`. Table 6 summarises and compares these assertion functions, with the assertion methods proposed in [131]. The experimental results for several case studies have confirmed the effectiveness of the proposed approach on debugging as well as on improving the success rate for quantum algorithms such as Quantum Fourier Transform (QFT) [64], Quantum Phase Estimation (QPE) [149], and Bernstein-Vazirani algorithm [29, 30].

The assertion schemes proposed by Huang and Martonosi [132], Zhou and Byrd [309], and Liu *et al.* [172], however, still have some limitations that they can only handle limited quantum states related assertions and limited assertion locations, which may increase the difficulty in debugging based on the assertions. To overcome this problem, Li *et al.* [168] proposed *Poq*, a projection-based runtime assertion checker for debugging quantum programs. *Poq* uses projection [37] to represent assertions, which, compared to classical representation, has more expressive power. Moreover, since projection naturally matches the projective measurement, which may not affect the measured state when the state is in one of its basis states [168], it may reduce the testing overhead. *Poq* can assert sophisticated quantum algorithms.

The above work on tackling bugs in quantum programs via assertion checking is promising, but it usually has to check assertions dynamically during runtime, which could waste the quantum computing (or simulation) resources. To overcome this, inspired by previous work on the verification of quantum programs such as quantum weakest preconditions [73] and quantum Floyd-Hoare logic [300], Singhal [258] proposed to encode assertions into a static type system of a quantum programming language to support programmers to write correct programs from the start. In this way, it is hopeful that the programmers can encode some of the semantic properties that they intend for their programs as specifications in their code, so that a type checker could be used to ensure the correctness of some of those properties during compilation time. The basic idea, to this end, is to extend the Hoare Type presented in Hoare Type Theory (HTT) in [199] for classical programming languages, to the quantum Hoare Types in the quantum domain which can be encoded into a quantum programming language to support static checking, and even formal verification of quantum software.

A Hoare type, similar to the Hoare triples and is represented as

$$\{P\}x : A\{Q\},$$

can be ascribed to a stateful computation if when executed in a heap satisfying the precondition P, the computation diverges or results in a heap satisfying the postcondition Q and returns a value of type A [199]. To support quantum circuits, the Hoare type is extended with its syntax augmented with that of QWire, a quantum circuit language that supports quantum circuit programming and verification. Through this way, the host language HTT can be augmented with a wire type W in QWire, so that to treat quantum circuits as data, it can be used to specify properties in quantum programs as well. Here, W and A can be described as:

$$W = 1 \mid bit \mid qbit \mid W1 \otimes W2$$

and

$$A ::= ... \mid Unit \mid Bool \mid A \times A \mid Circuit(W1, W2),$$

respectively. This work aims to build a unified system eventually for supporting quantum programming, specification, and verification, but it is still preliminary recently, and more work remains to be done to realize the goal.

7.4.3 **Debugging Quantum Processes.** Since the measurement of a quantum system may cause its state to collapse, it is challenging to debug quantum programs through the monitoring of the program. To solve this problem, Li and Ying [169] proposed an approach to debugging quantum processes through monitoring measurements. The basic idea of the approach is to develop a protocol to support the debugging of quantum processes through monitoring, which does not disturb the states of the processes.

Informally, the approach can be described as following. Suppose that we have built a quantum system to execute some quantum processes. The system is set to an initial state $|\psi\rangle$, and then evolve under the controlled Hamiltonian denoted by $H(t)$. This guarantees that the trajectory $\{|\psi\rangle\}$ of the system states is anticipated. Since the time for the whole process is usually much longer than the time of a single quantum component (such as a gate) action, it may be considered as infinite. Here, we can see that if there is a bug of the system in the process at time t', the system will not truly be $H(t)$ for $t \geq t'$ during the execution, which may cause errors in the system state, so we write ρ_t. For this situation, to debug the process, we should try to find a projection operator $P = 0$ of the system as well as a sequence of time points $t_1, t_2, \ldots (t_n \rightarrow \infty)$, such that $P|\psi_{t_n}\rangle = 0$ for all n. Here, $P|\psi_t\rangle = 0$ means that nothing can be detected by P if the system state is $|\psi_t\rangle$ as anticipated. So, to perform debugging, we can monitor the process at time $t_1, t_2, \ldots$, by using a measurement apparatus formalized by P, which is called a *monitoring measurement*, and together with probability $tr(P_{\rho_{t_n}})$, the error state could be detected at time t_n. If this really happened, an error would be detected in the state, and t' is more likely in $[t_{n-1}, t_n]$, and the relevant components should be carefully checked. Practically, the time points $t_1, t_2, \ldots$ are determined by a classical program S. The critical point for this debugging approach is to find the required projector P, and the condition $P|\psi_t\rangle = 0$ guarantees that the anticipated process is not disturbed by P. On the other hand, it is also implied that the debugging procedure is conclusive in the sense that if the process runs correctly, it would be no error

reported. Besides this, the authors also gave a formal definition of the proposed debugging approach in the case of discrete-time evolution.

As mentioned in the conclusion of the paper, the proposed debugging approach can only handle the debugging of the quantum processes with time-independent Hamiltonians, and the debugging with time-dependent hamiltonians is still an open problem [169].

7.4.4 **Language Support for Debugging.** Programming languages with good design features, such as static type systems and modular programming, can help prevent bugs. Hietala *et al.* [126] proposed a type-based approach to tracking errors in quantum programs at the language level. The approach uses the principle of quantum error correction code [110] for encoding each logic qubit in the program with a block of physical qubits, and then uses fault-tolerant operations to manipulate those encoded qubits to track and correct the errors in the programs. The goal is to determine whether the error correction operations added can successfully offset potential errors. To achieve this goal, it needs to track how errors propagate through the circuit and ensure that the cumulative number of errors in each encoded qubit does not exceed the threshold. To support all these issues, the authors extended QWIRE [217] quantum circuit language with new error types and an error correction gate EC by interpreting QWIRE's `Qubit` type as an encoded qubit and assuming that all gates are fault-tolerant. Since QWIRE is seamlessly integrated into the Coq proof assistant [31, 72], the language can utilize both QWIRE and Coq type system to check the errors in the circuits of the program.

A simple case study for the implementation was carried out and showed that the proposed language system could type check a teleport example correctly. This guaranteed that every function produces the specified number of errors in the given number of specified errors from the input, and the error threshold is never passed. Several issues are also discussed as the future work, including (1) a possible extension of the current type system with probabilities to check if all qubits have less than t (some threshold) errors with probability at least p, and (2) the development of a type system which the qubit type representing the physical qubits, rather than the encoded qubits, and the error is tracked through fidelity.

7.5 Quantum Program Analysis for Bug Detection

Program analysis utilizes static techniques for reliable computation of the information about the dynamic behavior of programs [203]. Example applications include compilers (for code improvement) and software validation (for detecting errors). Several researches have been carried out for program analysis of quantum programs recently, which can support bug detection for quantum programs.

ScaffCC [137, 138] is a scalable compiler framework developed at Princeton University for the quantum programming language Scaffold [1]. ScaffCC supports both compilation and compiling-time analysis for Scaffold programs to optimize and detect possible errors in the code. One analysis that ScaffCC supports is called *entanglement analysis*, which can conservatively identify each possible pair of qubits that might be entangled in the program. Such entangle information can help a programmer to design algorithms and to

debug. To perform entanglement analysis, ScaffCC explores dataflow analysis techniques to automatically track the entanglements within the code through annotating the output of the QASM-HL program, an intermediate representation of ScaffCC, to denote possibly entangled qubits. The analysis is conservative in the sense that it assumes that if two qubits interact, they are likely to have become entangled with each other. However, this might lead to some false positive entangled qubit pairs when the number of the qubits is large. Moreover, entanglement analysis can also help to find the un-computing portions in the module, through analysis of un-entanglement, which can be created by applying inverse functions such as CNOT and Toffoli operations to the same set of control and target qubits.

QASM [212] is an open-source Python tool that allows programmers to investigate how to map arbitrary computation onto a quadratic unconstrained binary optimization (QUBO) [288] that runs on a quantum annealing D-Wave 2X system [134]. QASM computes a simple heuristic computation algorithm and present symbolic variable names as well as assembly like code blocks that might be reusable. Developers may integrate the solution into the D-wave system directly or by using a local simulator, which raises questions about the reliability security of such a tool [8]. Alsaadi *et al.* [9] proposed to use static code analysis techniques to investigate the security threats in QASM code written in Python. The result indicates that the D-Wave system might be vulnerable to multiple security breaches. They used flow-sensitive, inter-procedural, and context-sensitive data flow analysis to uncover vulnerable points in the program, and several security threats have been found. They found that most of the identified functions and modules, though well structured, are either undefined or unused. Also, it is hard to apply fixes because QASM is highly dependent on D-WaveâĂŹs proprietary SAPI library (programmed in C++) and does not work correctly without it.

8 QUANTUM SOFTWARE MAINTENANCE

Support for software maintenance is crucially important in the development of computer software. A motivation for this is the well-known fact that somewhere between 60% and 80% of the cost of software comes from the maintenance phase of software development [285]. The primary purpose of quantum software maintenance is to modify and update software applications after delivery to correct faults and to improve performance. Research work on this area [159, 218] is emerging recently, which mainly focuses on the reengineering of existing classical software systems to integrate with new quantum algorithms.

8.1 Reengineering Classical Information Systems to Quantum computing

Software Reengineering [51, 69] refers to the inspection and modification of the target software system with a series of activities such as design recovery, re-documentation, reverse engineering, and forward engineering. It aims to reconstruct the existing system as a new form to develop higher quality and more maintainable software.

Although significant progress has been made in the development of quantum computers, it is evident that quantum computers can

not be used for all things in the short term (due to its high initial cost, among other reasons). Instead, it is common to use quantum computers to solve some important problems by making specific calls from classical computers to remote quantum computers in the cloud [268]. In this case, most enterprises require to integrate and migrate their first quantum algorithm or future quantum algorithm with their existing classical information systems. Therefore, reengineering must be revisited to deal with the problems associated with the migrations of quantum computing and the next coexistence of classical and quantum software.

To solve this problem, Pérez-Castillo [218] proposed a software modernization approach (model-driven reengineering) [243] to restructuring classical systems together with existing or new quantum algorithms to provide target systems that combine both classical and quantum information systems. The software modernization method in classical software engineering has been proved to be an effective mechanism that can realize the migration and evolution of software while retaining business knowledge. The solution proposed is systematic and based on existing, well-known standards such as Unified Modelling Language (UML) [40, 135, 240] and Knowledge Discovery Metamodel (KDM) [219]. The solution could benefit from reducing the development of new quantum information systems. Moreover, since it is based on international standards to represent the knowledge in an agnostic manner, it would be independent of any quantum programming languages. Besides, Pérez-Castillo also pointed out that quantum technologies and programming have not yet been addressed with techniques, good practices, and development methodologies of software engineering to meet quantum programs' needs.

Another issue regarding reengineering (maintenance) is how to integrate quantum computation primitives (for instance, quantum software components) to an existing classical software system. Since quantum computer (QC) is very different from the previous technology and method, integrating QC into existing software systems requires that it not only solves this problem at the level of algorithm implementation, but also involves many more extensive problems studied in software engineering [226]. Quantum annealing [89, 142, 254] is a quantum-mechanical metaheuristic (generic and approximate method) to solve combinatorial optimization and sampling problems. Krüger and Mauerer [159] performed a case study on how to augment a quantum software component implemented in a quantum annealer (QA) for the Boolean satisfiability problem (SAT) [63], to an existing software system. In this case study, they discussed the related quality measures of quantum components, and showed that the method of transforming the SAT into a QA, which is mathematically equivalent but structurally different, can lead to substantial differences in these qualities. Their research also showed that defining and testing specific quality attributes of QC components, as studied in [260], is one of the key challenges. Although these properties do not play a core role in classical engineering, they must be considered in the software architecture with quantum components. Their study may help readers form a realistic intuition about the ability, potential, and challenges of using quantum components to enhance classical software in the near and medium term. They also claimed that in the current development stage, this problem must be considered at a much lower level than the conventional abstraction level in classical software engineering.

The research also implies that the ability to easily and smoothly replace functional components of classical software architecture with quantum components, just like in classical component-based software engineering [124], is crucial for the success of the project.

9 QUANTUM SOFTWARE REUSE

Computer software can be systematically reused across the entire development life-cycle, i.e., requirement specification, design, and implementation [94, 158]. It has its place even in the post-delivery stages of development, e.g., its continuing quality assessment or software maintenance. This section gives an overview of state of the art on quantum software reuse in several aspects, including *quantum pattern reuse*, *quantum circuit reuse*, and *quantum state reuse*.

9.1 Quantum Pattern Reuse

As we discussed in Section 5, Leymann [166] identified some quantum patterns which can help develop quantum algorithms from the perspective of software reuse. They also plan to represent these quantum patterns in a pattern repository, which is a specialized database that stores pattern documents and manages the links between them. Such a quantum pattern repository allows a quantum software developer to query the database to find appropriate quantum patterns (e.g., to determine the entry pattern corresponding to a problem), supports browsing the content of each pattern document, and enables navigating between patterns based on the links between them. In this way, a quantum software developer could find suitable patterns from the repository, that may cross to several different domains, and compose them to solve a complex problem.

9.2 Quantum Circuit Reuse

The design of efficient quantum circuits is an essential problem in quantum computation [221, 293]. For a given unitary matrix, it is a difficult task to find a highly optimized quantum circuit. There are a few known methods of quantum circuit design, which are mainly based on heuristic search techniques such as genetic algorithm and simulated annealing. However, these methods are limited to a relatively small circuit size, and the solutions generated by the methods are usually difficult to explain. To solve these problems, Klappenecker and Rötteler [150] presented a new design principle for quantum circuits that is based exactly on the idea of reusing some existing quantum circuits in the construction of other quantum circuits. The unique characteristics of the design principle is: assuming that we have an effective set of quantum circuits available, we can systematically construct efficient quantum circuits by reusing and combining a group of highly optimized quantum circuits. The sound mathematical basis of this design method allows meaningful and natural explanations of the generated circuits. They also suggested that from a practical perspective, it would be interesting to build a database of medium-sized matrix groups with effective quantum circuits. A given transformation can then be searched in this database by linear algebra, and automatically deriving quantum circuit implementations in this way could be an attractive possibility.

Allouche *et al.* [7] attempted to extend further the design-by-reuse method proposed by Klappenecker and Rötteler [150] to a

general framework to support circuit synthesis through reusing the existing quantum circuits. In their extension, the approach needs to find suitable groups for the implementation of new quantum circuits. They also identified some critical points which are necessary for constructing their extended method. For example, this method relies on the distribution of the information between the group and the coefficient matrix. When the group contains enough information, such as the Fourier transform power group, the coefficient matrix is easy to calculate, and the efficiency of the quantum Fourier transform synthesis is used to generate an effective circuit on non-trivial operators. Besides, they also studied some potential group candidates, such as the *projective Pauli group* and the *dihedral group*, for testing the proposed method.

9.3 Quantum State Reuse

"*Some quantum state are hard to create and maintain, but are a valuable resource for computing. Twenty-first-century entrepreneurs could make a fortune selling disposable quantum states* [224]."

To make it real, Preskill presented an interesting idea called *plug-in quantum software* for probably reusing quantum states, which make up a quantum software program [224]. The basic idea is that manufacturers can design a valuable quantum state and use a special-purpose device to make multiple copies of that quantum state; these copies can be tested to ensure their quality and stored until they are needed. Consumers can pay to download that state and plug it into their quantum computer for a performance boost.

A candidate application of plug-in quantum software is to ensure the reliable operation of quantum computers, which is usually achieved by applying the principle of quantum error correction [256, 265]. For each known quantum error correction scheme, some quantum gates in the fault-tolerant universal set are easy to implement, while others are difficult. The latter can be efficiently executed with quantum software, which can be prepared in advance and then consumed during the execution of the quantum gate. The advantage of using software rather than hardware to implement the gate is that one can verify that the software is ready according to specifications before use. If a problem is found with the software, it can be discarded or repaired. On the contrary, if the hardware has a severe failure during the execution of the quantum gate, it may be difficult to recover.

The idea for creating and preparing quantum states offline is not new. Instead, several researches [111, 153, 255] have been carried out. Among them, Gottesman and Chuang [111] proposed an interesting approach to preparing, using, and classifying quantum states. The goal of their approach is to make the design of fault-tolerant quantum computers more straightforward and methodical. The main idea is to use a generalized form of quantum teleportation, a simple technique to reduce the required resources to construct necessary quantum gates (states), such as Pauli gates (X, Y and Z), Cliff group, Toffoli gate, the $\pi/8$ gate (rotation about the Z-axis by an angle $\pi/4$), and the phase gate. Their construction relies on the ability to create some ancilla states, which is independent of the data being manipulated, and therefore can be prepared offline in advance. So they are valuable general-purpose quantum resources, perhaps a kind of "quantum software," as mentioned by Gottesman

and Chuang [111], and may be considered as commercial products that can be manufactured.

10 CHALLENGES AND OPPORTUNITIES

Quantum computing is an emerging field where new engineering methodology needs to be developed to address issues such as unforeseen changes in requirements, lack of expertise in software development and limited budgets that plague scientific software projects [253]. The interdisciplinary nature of the quantum computing leads to the complexity of the field. This section discusses the challenges and opportunities in the area of quantum software engineering.

10.1 Quantum Software Requirements Analysis

Quantum software requirements analysis may include two stages: *stakeholder requirements definition* and *system requirements definition*. A key issue for the stakeholder requirements definition is to establish the critical performance requirements so that it is possible to determine whether quantum technology provides enough benefits over classical technology. Similarly, the acceptability (size, cost, capability) of the user system must be established as early as possible to determine the feasibility of using quantum technology in the application under consideration. Still, the types of benefits that quantum systems may provide are not fully defined. Therefore, if the existing technology can meet the needs, especially considering that the technical risks of using quantum systems may be high, users may not consider quantum systems. However, research and expected efforts to utilize quantum systems in the coming years will likely provide the knowledge needed to define the needs of quantum systems properly so that they can be considered in the solution space. Therefore, in the process of defining stakeholder needs, it is necessary to ensure that quantum and non-quantum solutions can be adequately distinguished so as not to ignore quantum benefits.

The definition of system requirements is to transform the stakeholder requirements into a set of (technical) system requirements that the proposed solution must meet. The resulting system requirements cover functional, non-functional, performance, process, and interface aspects, and include design constraints. However, due to the limitations of the models, the definition of design constraints due to quantum technology effects might also be problematic. Therefore, one of the main problems in the system requirements definition process is whether the models used to establish the relationship and correlation between different technical requirements are adequate and sufficient. This problem, however, can only be solved by developing better models and considering the effects of quantum technology adequately.

10.2 Quantum Software Design

This section discusses challenges and opportunities regarding quantum software design from the perspectives of *architectural design* and *detailed design*.

10.2.1 **Quantum Architectural Design.** The software architecture of a system defines its high-level structure, revealing its overall organization as a set of interacting components [179, 222].

A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction. The incorporation of quantum computing effects on software architectures may require new methods to define the attributes, features, functions, and constraints assigned to architectural elements. This will require quantum software architects to reach a consensus and establish standard representations for quantum software components. The extent to which new types of components are introduced at the architecture level depends on the level of abstraction required, which is yet unclear. The functional architecture is unlikely to be affected by the introduction of quantum software components, though specific functions within the architecture may be new.

Quantum Architectural Patterns (Styles). The software architecture pattern provides a skeleton or template for the overall software architecture or high-level design of software applications [109]. Shaw and Garlan [179] mentioned the architectural style (or pattern) of software architecture, which is the repetitive architecture used in various software applications (see also [22]). Architectural patterns can be classified into two main categories: *architectural structure patterns*, which focus on the static structure of the architecture, and *architectural communication patterns*, which focus on the dynamic communication among distributed components of the architecture. Currently, software architects do not have architectural patterns for quantum software systems, which can be used to adequately deal with the quantum issues when specifying the architecture of quantum software systems. The current repository for architectural patterns do not provide patterns which consider the quantum issues. Therefore, to perform the architectural-level design of quantum systems, it is necessary to identify some *quantum architectural patterns*, which are well-considered to model the quantum effect of the system.

Quantum Architectural Description Languages. Architectural description languages (ADLs) [59] are formal languages that can be used to represent the architecture of a software system. They focus on the high-level structure of the overall application rather than the implementation details of any specific source module. Several ADLs have been proposed, such as Wright [235], Rapide [176], UniCon [252], Darwin [177], and ACME [99] to support formal representation and reasoning of classical software architectures. Recently, software architects do not have tools to adequately deal with the quantum issues when specifying the architecture of quantum software systems. Current ADLs do not provide primitives to specify quantum components and their connections with classical components. Using ADLs, software architects can specify the functionality of the system using components and the interaction between components using connectors. It is necessary to extend current ADLs to *quantum architectural description languages (qADLs)* to formally specify the architectures of quantum software systems for architectural-level design of quantum systems. We believe that such a qADL should contain at least those mechanisms, including specifications of classical components with interfaces and connections between interfaces (already provided in classical ADLs), specifications of quantum components, and specifications of connectors between classical and quantum components. Also, a qADL should support the tasks of formal analysis, verification, and

validation of quantum software architectures.

Software Quality Attributes. Software quality attributes [22] refer to the non-functional requirements of software, which can have a profound effect on the quality of a software product. The software quality attributes of a system should be considered and evaluated during the development of the software architecture. These attributes relate to how the architecture addresses important non-functional requirements, such as performance, security, and maintainability. Other software quality attributes include modifiability, testability, traceability, scalability, reusability, and availability. As we summarized in section 5, Sohdi [260] identified some critical characteristics of quantum computing systems, and studied how these characteristics may affect the quality attributes of quantum software architectures. A more significant issue worth investigating is how to use these quality attributes to evaluate the quality of the quantum software architecture during the architectural design.

10.2.2 **Detailed Quantum Design.** The detailed quantum design provides information about the quantum software system as a whole and the individual quantum software components to enable implementation. An essential aspect of the detailed design is the selection of the technologies required for each quantum software component. Substantially, the inclusion of quantum software component increases the search space available to the quantum software engineer in making technology selection, but may lead to a challenge. The development of suitable models and their incorporation into quantum software design frameworks should be an area of intensive research effort in the future. Another important consideration is that at any particular stage in the quantum software life cycle, an appropriate level of quantum software modeling must be included. Moreover, it is crucial to recognize that the individual quantum software components may be developed by several or many different organizations.

Quantum Design Patterns. A design pattern describes a recurring design problem to be solved, a solution to the problem, and the context in which that solution works [95, 97]. Design patterns have been evolved over a long period and provide the best solutions to specific problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way. Design patterns have proved highly useful within the object-oriented field and have helped to achieve good design of applications through the reusability of validated components. These experiences tell us that it is crucial to identify the elements of good and reusable designs for quantum software and to start formalizing peopleâĂŹs experience with these designs through quantum design patterns. We hope that the quantum design patterns, if being identified, could benefit the quantum software development, and a quantum pattern catalog for quantum software could be especially useful.

10.2.3 **Design Models for Quantum Software.** There is an absence of modeling techniques for quantum software system design. The current existing model of quantum software systems [220] is generally a simple extension of the classical modeling technique. The lack of suitable models is probably one of the most significant difficulties facing quantum software engineering, particularly as

this may impact design, testing, and possibly maintenance parts of the quantum software life cycle.

Design models for classical software engineering have been extensively studied from various perspectives. Among them, several notable design models are *data flow diagrams* (DFDs) [305], *entity-relationship diagrams* (ERDs) [49] and *unified modeling language* (UML) [40]. We believe that the first natural step is to investigate these well-established design models to see if they could be extended to the quantum computing domain to support quantum software modeling by incorporating the quantum effects within the models appropriately.

10.3 Quantum Software Implementation

Quantum software implementation refers to the development of quantum software entities that meet the requirements, architecture, and design. It is not uncommon to introduce new constraints in the implementation. These constraints must be reflected back into requirements. The essential design characteristics for achieving a subtle quantum effect are likely not to be compromised. Therefore, the process must include rules and tests to ensure that unwanted changes are not introduced. One can expect that reliability will become an important area of research in quantum software technologies. The principles derived from this research will provide a basis for the implementation strategy formulated during the implementation process. In short, the implementation process will need to incorporate the principles of quantum software reliability engineering. Testing the potential impact of implementation changes on the entire quantum software system requires modeling before agreeing to the changes.

On the other hand, it seems challenging to introduce new quantum programming languages into widespread practice. Perhaps a promising way is to define the requirements for the future high-level quantum programming, which may eventually lead to the development and widespread use of a more efficient quantum programming language. Another possible research direction could be to develop techniques and tools to generate quantum code from quantum software design specifications automatically.

10.4 Quantum Software Reliability

This section discusses the challenges and opportunities on quantum software reliability, including fault model, testing, debugging, verification, and visualization.

10.4.1 **Fault Model for Quantum Software.** In general, a fault model refers to an engineering model that may cause problems in the construction or operation of equipment, structures, or software [36]. The unique features of quantum programming, such as superposition, entanglement, and no-cloning, do not occur in classical imperative, functional, or even multi-paradigm programming. Each feature can manifest new fault types that ultimately lead to quantum program failures. Therefore, to test quantum software, a fault model for quantum software is required. Such a fault model should be based on the peculiarities of quantum programs, and defined through careful analysis of the unique constructs of quantum programming languages and reflects an initial evaluation of classes of potential faults. Such a fault model can be used to measure the fault-detection effectiveness of automatic test generation and

selection techniques of quantum software. Although work on identifying bug types for quantum software is just beginning [132, 133], more study should be carried out to build a practical fault model for supporting quantum software testing and debugging.

10.4.2 **Quantum Software Testing.** Systematic testing of quantum software systems must be based on fault models that reflect the structural and behavioral characteristics of quantum software. Criteria and strategies for testing quantum programs should be developed in terms of the fault model. As its classical counterpart, quantum software testing must explore the following issues rigorously and provide the answers for each of them to build effective and efficient testing frameworks for quantum software.

(1) How to define testing coverage criteria of quantum software?
(2) How to automatically and efficiently generate test cases for quantum software?
(3) How to evaluate the test data quality for quantum software?
(4) How to test quantum software regressively?

Although some of these issues, such as testing covering criteria [289] and test case generation [131], have been addressed in current researches, it is far from practical uses for testing quantum software.

10.4.3 **Quantum Program Debugging.** A commonly used classical debugging technique is the one in which a developer examines the program state through setting breakpoints [164]. This technique, however, cannot be used to debug quantum software since any inspection of a quantum register can cause it to decohere [295]. A simple variant of this technique that entails making a copies of the quantum register is similarly foiled due to the physical impossibility of making copies of quantum objects. One possible way is to use multiple quantum registers that are prepared in the same way. However, the probabilistic nature of quantum computation should serve as a constant reminder to the debugger of quantum software that no two quantum registers can be assumed to be identical. It is evident here that more investigations are needed. Recently, several pieces of research [132, 133, 168, 169, 172, 309] have been carried out that showed some promising initial results on debugging quantum software. However, it is still not clear what the appropriate debugging techniques for quantum computing are [195]. As a result, new approaches still need to be developed. On the other hand, while assertion-based debugging [132, 133, 168, 172] seems a promising way in debugging quantum software, different kinds of debugging techniques in classical computing, such as interactive debugging, static and dynamic analysis, code review and inspection, and postmortem, are also worth further investigation [195]. A question that naturally arises is: *are these classical debugging techniques applicable to the quantum domain?*

As an example, *Whyline* [155], a novel interactive debugging paradigm for classical software, can reduce the debugging time by allowing programmers to select "why" and "why not" questions to query the behavior of a program. We have not seen such a novel idea in the debugging of quantum software, but it would be worth exploring. An in-depth understanding of how quantum programmers perform debugging would provide an opportunity to develop new quantum debugging tools, and such work should

be based on previous research that demonstrates best practices in scientific computing [16].

Another example of classical debugging paradigms that should be paid attention to is *algorithmic debugging* [250, 251]. Algorithmic debugging is an interactive process in which the debugging system gains knowledge of the expected behavior of the program being debugged and uses this knowledge to locate errors. An algorithmic debugger can be invoked after noticing an externally visible symptom of a bug. Then, it executes the program and builds an execution trace tree at the procedure level, while saving some useful trace information such as the procedure name and input/output parameter values. After that, the debugger traverses the execution tree and interacts with the user by asking for the expected behavior of each procedure. The user can answer "yes" or "no" to give an assertion about the predicted behavior of the procedure. Once some necessary conditions are satisfied, the search ends, and the location of the bug is identified. Algorithmic debugging provides a declarative way of debugging. If applied to quantum software, it might offer abstractions that allow programmers to think at an algorithmic level with less concern for details like control pulse generation, which may provide a possibility for overcoming the problems for which the quantum software debugging are facing [54].

10.4.4 **Quantum Software Visualization.** Understanding the behavior of quantum software is an important and nontrivial task, and software visualization [151, 264], a well-developed technique for understanding the behavior of classical software, may significantly contribute to it. Visualization of qubit states is particularly challenging since the number of achievable quantum states is exponential [16]. Recently, several methods have been proposed for visualizing the state of qubits by using Bloch sphere [62, 106] and matrices of Pauli Expectation Value (PEVs) [117, 294], and for visualizing the transitions between qubit states by using two-qubit representation [233]. However, the explosion of qubit state-space necessitates the development of scalable visualization that can intuitively help quantum software developers understand the states their systems are in, to facilitate the development, debugging, and validation of quantum algorithms [16].

10.4.5 **Quantum Program Verification.** Verification plays an essential role in quantum programming, both in the short and long term. Quantum programs are difficult to debug and test due to features such as superposition, entanglement, and no-cloning. To build the confidence regarding the correctness of a quantum program, we need to develop verification methods. There has been substantial progress in verifying quantum programs [43, 47, 84, 173, 228, 229, 300, 304]. However, as Rand *et al.* [229] pointed out, novel verification methods are still needed to deal with errors and to verify error-prone quantum programs concerning the hardware we intend to run them on. Moreover, we need also approaches to verify quantum compilations [127].

10.5   Quantum Software Maintenance

Software maintenance is an essential activity during software development. The objective is to modify the existing software product while preserving its integrity [27]. Methods, techniques, and tools for classical software maintenance have been well studied and established [27, 75, 119, 262], but research on the maintenance of quantum software systems is just starting [218]. We believe that any quantum software maintenance process should deal with at least the following three main issues:

- How to understand the existing quantum software?
- How to modify the existing quantum software?
- How to re-validate the modified quantum software?

Moreover, the maintenance process of quantum software systems likely needs to include the monitoring of the systems so that fault diagnosis could be informed during the maintenance and evolution of the systems.

10.6   Quantum Software Reuse

Component-based software engineering (CBSE) is an approach to software systems development based on reusing software components [124, 156]. Component here means a unit or a part of a model. It is the process that emphasizes the design and construction of computer-based systems using reusable software components or simply building software systems from pre-existing components [262]. The maintenance of CBSE can be easily accomplished by replacing or introducing a new component to the system [183].

As quantum software resources are getting accumulated, it is crucial to develop some methodologies, techniques, and tools for reusing quantum software systems. One promising way, we believe, might be the component-based quantum software engineering (CBQSE), which focuses on the design and development of quantum computer-based systems with the use of reusable quantum (classical) software components. Such an approach can save the development time and also be easy for maintenance during the evolution of the quantum software system. Another possible research direction, as we discussed in Section 10.2, is to build a quantum architectural (design) pattern catalog to support the reuse of quantum software entities efficiently and effectively.

11   RELATED WORK

To the best of our knowledge, this work is the first comprehensive survey on the research of quantum software engineering regarding various phases of its life cycle, including quantum software requirements analysis, design, implementation, testing, and maintenance. This section discusses some related work in quantum programming languages, quantum software engineering, and quantum software development environments.

11.1   Quantum Programming Languages

Software is a solution for solving a computational problem using a formal programming language. The construction of the language and the tools that can be used to model, implement, and test software systems affect the quality of the solution, including correctness, reliability, readability, computational efficiency, and efficiency of design and development. Therefore, the quantum programming language plays a vital role in the life cycle of quantum software development. Several comprehensive surveys on quantum programming languages have been carried out from different perspectives [54, 98, 102, 125, 140, 186, 239, 244, 261, 263, 276, 280, 281, 302, 311].

Selinger [244] carried out the first survey on quantum programming language in 2004. His survey uses a similar classification scheme and offers a different perspective on some of the issues in the field.

Gay [102] published a comprehensive and detailed survey on quantum programming languages in 2006. He classified the central theme of each paper from the perspectives of programming language design, semantics, and compilation. Regarding to programming language design, the paper considers imperative and functional programming languages, and for semantics, it focuses on the denotational techniques. Besides the survey, Gay also maintained an online bibliography as a resource for the community [100].

Unruh [276] also gave an overview of the state of the art on the development of quantum programming languages in 2006. In his overview, quantum programming languages are classified into two types, i.e., *practical programming language* and *formal programming language*. Practical programming languages, such as QCL [206, 207] and Q [33], aim at practical applications such as simulation or the programming of actual quantum computers, while formal programming languages, such as QFC (QPL) [244, 245, 247], Quantum lambda calculus [181], qGCL [241, 312–314], QPAlg [160], and CQP [103, 104], concentrate on how to model the semantics of a quantum program. The survey also pointed out some open problems and challenges regarding the practical and formal programming languages. For practical languages, the main challenges are how to design some powerful language constructs to abstract from the low-level model of quantum circuits to lift towards the high-level programming, and how to develop compilers and optimizers to obtain the best results out of the available resources. For formal languages, the main challenges lay in how to design expressive, easy-to-read quantum languages with well-defined semantics, and how to develop effective methods to verify quantum programs written by these languages.

Rüdiger [239] presented a comprehensive survey for quantum programming languages from the pragmatic perspective in 2006. He first gave some introduction on necessary notations of quantum theory and quantum computation, and then discussed the goals and design issues for quantum programming languages. He claimed that quantum programming languages should enable programmers to reason about the structures of quantum algorithms and programs, and a well-designed quantum programming language should help find new quantum algorithms. The survey also discussed several concrete quantum programming languages such as pseudocode [152, 154], QCL [206, 207], and Q language [32, 33] in detail, and gave a comparison of these three languages from the perspective of language paradigms and semantics. The survey also pointed out some research directions towards quantum programming language design and implementation.

Jorrand's survey [140] on the quantum computing paradigm started from a brief overview of the principles that underlay quantum computing and introduced the breakthroughs achieved by the quantum computing field, which include quantum algorithms and teleportation. The main focus of the survey is on the quantum computation paradigm, that is, quantum programming languages and their semantics. Three styles for quantum programming languages have been discussed in the survey, including *imperative*, *functional*, and *parallel and distributed*. For each style, several

languages are mentioned and compared, that is, QCL [206, 207] and qGCL [241, 312–314] for imperative, the approaches by Tonder [282] and Girard [107] and QPL [244, 245, 247] by Selinger for functional, and CQP [103, 104] and QPAlg [160] for parallel and distributed. The survey also discussed the semantics issues of quantum programming languages, from the perspectives of operational, axiomatic, and denotational aspects, respectively.

Sofge [261] gave a brief survey on quantum programming languages from the perspective of history, method, and tools in 2008. He introduced a taxonomy of quantum programming languages, which divided into (1) imperative quantum programming languages, (2) functional quantum programming languages, and (3) other quantum programming language paradigms, which include some types of mathematical formalism that not intended for computer execution. Based on the taxonomy, a brief survey is given to reflect state of the art for quantum programming languages until 2007. Some challenges and a path forward for quantum programming languages are also given in the survey.

Ying *et al.* [302] presented a survey on the programming methodology for quantum computing in 2012. Their survey discussed those issues regarding the design of sequential and concurrent quantum programming languages and their semantics and implementations, with more emphasis on the work conducted by the authors. Notably, in addition to discussion on the quantum programming languages themselves, they also reviewed some formal verification approaches for quantum programs and protocols, and discussed the potential applications of these quantum languages and verification approaches in quantum computing engineering.

One of the most recent surveys for quantum programming languages is proposed by Garhwal *et al.* [98] in 2019, which gave an overview of state of the art in the field of quantum programming languages, focusing on actual high-level quantum programming languages, their features and comparisons. They classified quantum programming languages into five categories: quantum imperative programming language, quantum functional language, quantum circuit language, multi-paradigm language, and quantum object-oriented language. In the survey, they tried to answer the following research questions [98]:

- What are the different types of quantum languages?
- What are the recent trends in developing quantum languages?
- What are the most popular publication venues for quantum programming?
- What are the most cited papers in the area of a quantum language?
- Which major companies, groups, institutes, and universities are working on the development of quantum languages?

They concluded the survey by pointing out some recent trends, results, and future work in the area.

Another recent survey [311] is presented by Zorzi, which focuses on the QRAM (Quantum Random Access Machine) architectural model. The survey is organized from the twofold perspectives, i.e., theory and concrete, and identifies the main problems one has to face in quantum language design. The survey tried to find some evidences from current researches to answer the following fundamental questions [311] regarding quantum language design and provided some possible answers to these questions.

- What is the architectural model the language refers to?
- How to manage quantum data (which are no-duplicable to ensure the no-cloning property, so need a form of linear treatment)?
- What class of quantum functions one aim to program (expressive power)?

Besides those above, several other surveys are also given in [54, 125, 186, 263, 280, 281], on quantum programming languages from different viewpoints. However, although these surveys on quantum programming languages give very comprehensive pictures on the state of the art of researches on quantum programming languages, they lack the discussion of other phases of the life cycle of quantum software development, such as requirement, design, testing, and maintenance.

11.2   Quantum Software Engineering

The term *quantum software engineering* was originally coined by John Clark and Susan Stepney [57] at the Workshop on Grand Challenges for Computing Research organized by Tony Hoare and Malcolm Atkinson, in 2002. Since then, there have been extensive research works on the different aspects of quantum software engineering as surveyed in this paper. Although no survey has been presented for quantum software engineering until now, some papers [19, 223] discussed the challenges of which quantum computing is facing on during quantum software development.

11.2.1   **Journeys in Non-Classical Computation.** In their seminal papers titled "Journeys in non-classical computation I and II" [267, 268], Stepney *et al.* presented a grand challenge on quantum software engineering, to develop a mature discipline of quantum software engineering for fully exploiting commercial quantum computer hardware. They claimed in [268] that "*The whole of classical software engineering needs to be reworked and extended into the quantum domain*.", and introduced this challenge from different perspectives of quantum software engineering, including foundations, quantum computational models, languages and compilers, methods and tools, as well as novel quantum possibilities. Here, we briefly list some challenge problems for each aspect mentioned in [267, 268]:

- *Foundations*: how to develop metaphors and models of quantum computing, which one can use to design and reason about quantum algorithms without considering the details within the quantum machine and unitary matrices, etc.
- *Quantum computational models*: how to generalize various classical formalisms to the quantum realm in different ways
- *Languages and compilers*: how to determine the fundamental building blocks of quantum programming; how to design suitable assembly level and high-level quantum programming languages and the compilers of these languages; how to develop suitable reasoning systems and refinement calculi for these languages.
- *Methods and tools*: how to develop powerful simulation systems so that one can perform computational experiments and validate the designs of the languages and algorithms; how to discover what high-level structuring techniques and architectures are suitable for quantum software; how to develop novel

debugging and testing techniques for quantum computing, as we know that quantum execution is in principle unobservable.
- *Novel quantum possibilities*: how to extend quantum software engineering to encompass those new issues from quantum mechanics, which cannot be even simulated by discrete deterministic classical computers.

We believe that the software engineering and the quantum computing communities should pay more attention to the issues listed above, to facilitate the construction of the overall picture for the field of quantum software engineering.

11.2.2   **A landscape of Research Challenges for Quantum Software Engineering.** Barbosa [19] presented a landscape of some research challenges to overcome for evolving the software engineering application in quantum computing, along with potential research directions in several aspects of the problem, including models, architectures, and properties. He believed that it is the time to discuss an agenda for a solid, rigorous software engineering discipline for quantum systems. He claimed that any roadmap for such a discipline should contain three main aspects:

(1) How quantum software systems are modeled.
(2) How the models of these systems are composed.
(3) How the properties of these systems' behaviors can be predicted, specified, and verified.

He also pointed out some challenges and research directions in terms of the models, architectures, and properties of the quantum systems, respectively. Among those directions, one particularly interesting issue, which is worth exploring, is how to extend the contract-based design, a successful paradigm in classical software engineering, to the quantum domain. The paper is presented in the form, of which the formal aspect of the quantum software engineering principle is emphasized.

11.2.3   **The Talavera Manifesto for Quantum Software Engineering and Programming.** Piattini *et al.* [223] presented the Talavera Manifesto for quantum software engineering and programming, which is the result obtaining from the discussions and different viewpoints of academia and industry practitioners who joined at the QANSWER, the 1st International Workshop on Quantum Software Engineering & Programming, promoted by aQuantum at School of Computer Science, in the Campus of Talavera de la Reina of the University of Castilla-La Mancha. The manifesto collects some principles and commitments about the field of quantum software engineering and programming, as well as some calls for action. They believe that quantum software engineering should have a necessary contribution to the success of quantum computing, and it is the time to develop quantum software by applying or adapting the well-established principles and methodologies from classical software engineering field, to the development of quantum software, which may include processes, methods, techniques, and practices. Concretely, they listed some principles and commitments regarding quantum software engineering, such as quantum software *process*, *reengineering*, *requirement*, *evolution*, *testing and debugging*, *reuse*, *security and privacy*, and *management*. They also call for actions to the stakeholders, including software practitioners, researchers, educators, government and funding agencies, quantum technology

vendors, professional associations, customers, and users, who may contribute to the field of quantum software engineering.

However, the manifesto contains no detailed discussion about the current status of research in the field of quantum software engineering.

11.3 Quantum Software Development Environments

Roetteler *et al.* [236] presented a survey on investigating recent progress in building a quantum software framework, which can compile quantum algorithms from the high-level descriptions to physical quantum gates, which can be implemented on fault-tolerant quantum computers. In the survey, they discussed why tools such as compilation and design automation are essential to meet the enormous challenges of building scalable quantum computers. They also introduced a library developed with LIQUi|⟩ [292] programming language, including reversible circuits for arithmetic and new real quantum methods that rely on quantum computer architecture, which allows the probability execution of quantum gates. This library, in some cases, can reduce time and space overhead. Also, the survey highlights why these libraries are useful for implementing many quantum algorithms. Finally, the tool Revs was investigated. This tool can help to compile high-level irreversible programs into low-level reversible circuits with resource efficiency, while trying to optimize the memory footprint of the resulting reversible network. Construction of the tool is motivated by the fact that the availability of qubit is limited in the foreseeable future.

LaRose [163] presented an overview and comparison of gate-level quantum software platforms. The overview mainly focuses on four new software platforms in quantum computing, namely, Forest (pyQuil) from Rigetti [61, 259], Qiskit from IBM [232], ProjectQ from ETH Zurich [266, 272], and Quantum Developer Kit (Q#) from Microsoft [269]. Forest, Qiskit, and ProjectQ allow the users to access the real quantum computing devices, while Quantum Developer Kit only allows access to the simulator. In the overview, each platform is discussed and summarized from six aspects: requirements and installation, documentation and tutorials, quantum programming language syntax, quantum assembly/instruction language, quantum hardware, and simulator capabilities, with the links to documents and tutorial sources of each package. The overview also compares each platform with additional aspects, including library support, quantum hardware, and quantum compilers, and lists some notable and useful features of each platform. The purpose of this overview is to provide users with essential information for each platform to help them select a suitable platform to start exploring quantum computing programming. In addition to the four platforms mentioned, the overview also briefly introduced several quantum programming environments, including Quipper [114, 115], Scaffold [1], and QCL [206].

Fingerhuth *et al.* [86] presented an exhaustive review of open-source software projects in quantum computing. The review covers the whole stages in the quantum toolchain from quantum hardware interfaces through quantum compiler to implementations of quantum algorithms and also the full spectrum of quantum computing paradigms, which include quantum annealing [89, 142, 254],

and discrete and continuous-variable gate-model quantum computing [210]. For each project, the evaluation covers those features, including documentation, licensing, the choice of programming language, compliance with software engineering specifications, and the culture of the project. Fingerhuth *et al.* also discussed some of the findings that though the diversity of the projects is fascinating, only a few projects attract external developers, and even many commercially supported frameworks have flaws in software engineering. Based on these findings, the authors highlighted some best practices that could foster a more active community around quantum computing software, welcome newcomers to the field, and also ensure high-quality, well-documented code.

Shaydulin *et al.* [253] surveyed open-source quantum computing projects from a different perspective, which mainly focused on the contributors of quantum software projects. They observed that one of the main problems in quantum computing is the lack of understanding of what training is required for success in the quantum computing field. To answer this question, they collected data on 148 contributors to three open-source quantum computing projects hosted on GitHub, including Qiskit [4] by IBM, PyQuil/Grove [259] by Rigetti, and Cirq [271] by Google. They studied the successful contributors to these projects from the field as well as the wider quantum computing community to understand the background and training that contributed to their success. These observations can help develop educational resources targeted at preparing a new generation of quantum computing researchers and practitioners. Their study could have a positive effect on bringing software engineering methodologies and techniques to the quantum domain.

All these surveys focus specifically on the quantum software development environments, one of the essential aspects of quantum software development, but not on the whole life cycle of quantum software development, as we presented in this paper.

12 CONCLUDING REMARKS

This paper presented a comprehensive survey in the field of quantum software engineering, which is across all the phases of the quantum software life cycle, and covered the crucial issue on quantum software reuse as well. The paper also discussed some challenges and opportunities in the field.

In its short history, quantum software development has been driven by the emergence of quantum programming languages [100, 115, 206, 232, 269]. In other words, quantum software development has mostly been synonymous with quantum programming. While such languages have helped popularise the subject, this is not a healthy position in the longer term. In particular, it is vital that a complete software engineering discipline emerges for quantum software development. This paper is aimed as a stepping stone in this direction. This paper has examined state of the art in engineering support for quantum software systems, looking at areas such as requirements, design, implementation, testing, and maintenance in the quantum software life cycle. The crucial issue of quantum software reuse has also been considered. The evidence presented suggests that rapid progress is being made in these areas. It would be wrong though to claim maturity in the topics addressed by this paper; indeed, although many techniques have emerged, further experience is required to appreciate their relative strengths

and indeed to consolidate proposals into a small number of critical approaches.

REFERENCES

[1] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. 2012. *Scaffold: Quantum programming language*. Technical Report. Department of Computer Science, Princeton University.

[2] Jean-Raymond Abrial, SA Schuman, and Bertrand Meyer. 1980. A Specification language. *RM McK-eag and AM MacNaughten, editors7 On the Construction of Programs: An Advanced Course* (1980).

[3] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. 1982. Validation, verification, and testing of computer software. *Comput. Surveys* 14, 2 (1982), 159–192.

[4] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-HernÃądnez, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. CÃşrcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente GonzÃąlez, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, ÅĄukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose MartÃ ̆n-FernÃądndez, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda RodrÃ ̆nguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, JesÃžs PÃ ̌rez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, AbdÃˇn RodrÃ ̆nguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan WÃűrner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An Open-source Framework for Quantum Computing. (jan 2019). https://doi.org/10.5281/zenodo.2562111

[5] Christopher Alexander. 1977. *A pattern language: towns, buildings, construction*. Oxford university press.

[6] Eric Allen. 2002. Bug Patterns in Java. (2002).

[7] Cyril Allouche, Marc Baboulin, T Goubault de Brugière, and Benoît Valiron. 2017. Reuse method for quantum circuit synthesis. In *International Conference on Applied Mathematics, Modeling and Computational Science*. Springer, 3–12.

[8] Khalifa AlRoum, Abdulhakim Alolama, Rami Kamel, May El Barachi, and Monther Aldwairi. 2017. Detecting malware domains: A cyber-threat alarm system. In *International Conference on Emerging Technologies for Developing Countries*. Springer, 181–191.

[9] Hesham H Alsaadi, Monther Aldwairi, and Eva-Marie Muller-Stuler. 2019. Analyzing D-Wave Quantum Macro Assembler Security. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*. Springer, 125–132.

[10] Thorsten Altenkirch and Jonathan Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE, 249–258.

[11] Thorsten Altenkirch and Jonathan Grattage. 2005. QML: Quantum data and control. *Submitted for publication, Febuary* (2005).

[12] Deepak Alur, John Crupi, and Dan Malks. 2003. *Core J2EE patterns: best practices and design strategies*. Prentice Hall Professional.

[13] Stephen J Andriole and Peter A Freeman. 1993. Software systems engineering: The case for a new discipline. *Software Engineering Journal* 8, 3 (1993), 165–179.

[14] Krzysztof R Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of HoareâĂŹs logic. *Formal Aspects of Computing* 31, 6 (2019), 751–807.

[15] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. 1991. A general framework for debugging. *IEEE software* 8, 3 (1991), 14–20.

[16] Zahra Ashktorab, Justin D Weisz, and Maryam Ashoori. 2019. Thinking Too Classically: Research Topics in Human-Quantum Computer Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[17] Harrison Ball, Michael J Biercuk, Andre Carvalho, Rajib Chakravorty, Jiayin Chen, Leonardo A de Castro, Steven Gore, David Hover, Michael Hush, Per J Liebermann, et al. 2020. Software tools for quantum control: Improving quantum computer performance through noise and error suppression. *arXiv preprint arXiv:2001.04060* (2020).

[18] Alexandru Baltag and Sonja Smets. 2006. LQP: the dynamic logic of quantum information. *Mathematical structures in computer science* 16, 3 (2006), 491–525.

[19] Luís S. Barbosa. 2020. Software engineering for 'quantum advantage'. In *2020 IEEE/ACM First International Workshop on Quantum Software Engineering (Q-SE)*. IEEE.

[20] Rami Barends, Julian Kelly, Anthony Megrant, Andrzej Veitia, Daniel Sank, Evan Jeffrey, Ted C White, Josh Mutus, Austin G Fowler, Brooks Campbell, et al. 2014. Superconducting quantum circuits at the surface code threshold for fault tolerance. *Nature* 508, 7497 (2014), 500–503.

[21] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software architecture in practice*. Addison-Wesley Professional.

[22] Len Bass, Mark Klein, and Felix Bachmann. 2000. *Quality attribute design primitives*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[23] K BECK. 1987. Using Pattern Languages for Object-Oriented Programs. In *OOPSLA-87*.

[24] B Beiser. 1983. *Software Testing Techniques*. Van Nostrand Reinhold Company.

[25] Marcello Benedetti, John Realpe-Gómez, Rupak Biswas, and Alejandro Perdomo-Ortiz. 2016. Estimation of effective temperatures in quantum annealers for sampling applications: A case study with possible applications in deep learning. *Physical Review A* 94, 2 (2016), 022308.

[26] Charles H Bennett and Gilles Brassard. 2014. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science* 560 (2014), 7–11.

[27] Keith H Bennett and Václav T Rajlich. 2000. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 73–87.

[28] Jon Bentley. 1985. Programming Pearls: Confessions of a Coder. *Commun. ACM* 28, 7 (1985), 671–679.

[29] Ethan Bernstein and Umesh Vazirani. 1993. Quantum complexity theory. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of Computing*. 11–20.

[30] Ethan Bernstein and Umesh Vazirani. 1997. Quantum complexity theory. *SIAM Journal on computing* 26, 5 (1997), 1411–1473.

[31] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: CoqâĂŹArt: the calculus of inductive constructions*. Springer Science & Business Media.

[32] Stefano Bettelli. February 2002. *Toward an architecture for quantum programming*. Ph.D. Dissertation. Università di Trento.

[33] Stefano Bettelli, Tommaso Calarco, and Luciano Serafini. 2003. Toward an architecture for quantum programming. *The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics* 25, 2 (2003), 181–200.

[34] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195–202.

[35] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[36] Robert Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

[37] Garrett Birkhoff and John Von Neumann. 1936. The logic of quantum mechanics. *Annals of mathematics* (1936), 823–843.

[38] Barry W. Boehm. 1976. Software Engineering. *IEEE transactions on computers* 25, 12 (1976), 1226–1240.

[39] Barry W Boehm. 1988. A spiral model of software development and enhancement. *Computer* 21, 5 (1988), 61–72.

[40] Grady Booch, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide*. addison-wesley.

[41] Pierre Bourque, Richard E Fairley, et al. 2014. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

[42] JH Bradley. 1985. The Science and Art of Debugging. *COMPUTERWORLD*. 19, 33 (1985), 35–38.

[43] Olivier Brunet and Philippe Jorrand. 2004. Dynamic quantum logic for quantum programs. *International Journal of Quantum Information* 2, 01 (2004), 45–54.

[44] Frank Buschmann, Kelvin Henney, and Douglas Schimdt. 2007. *Pattern-Oriented Software Architecture: On Patterns And Pattern Language, Volume 5*. Vol. 5. John wiley & sons.

[45] Carmelo R. Cartiere. 2013. *Quantum Software Engineering: Bringing the Classical Software Engineering into the Quantum Domain*. Master's thesis. Department of Computer Science, University of Oxford.

[46] Carmelo R. Cartiere. 2016. Quantum Software Engineering: Introducing Formal Methods into Quantum Computing.

[47] Rohit Chadha, Paulo Mateus, and Amílcar Sernadas. 2006. Reasoning about imperative quantum programs. *Electronic Notes in Theoretical Computer Science* 158 (2006), 19–39.

[48] Arnab Chakraborty. 2011. QuECT: a new quantum programming paradigm. *arXiv preprint arXiv:1104.0497* (2011).

[49] Peter Pin-Shan Chen. 1976. The entity-relationship modelâĂŤtoward a unified view of data. *ACM transactions on database systems (TODS)* 1, 1 (1976), 9–36.

[50] Harold Chestnut. 1967. Systems Engineering Methods. (1967).

[51] Elliot J. Chikofsky and James H Cross. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE software* 7, 1 (1990), 13–17.

[52] Andrew M Childs, Dmitri Maslov, Yunseong Nam, Neil J Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (2018), 9456–9461.

[53] Kenta Cho. 2016. Semantics for a quantum programming language by operator algebras. *New Generation Computing* 34, 1-2 (2016), 25–68.

[54] Frederic T Chong, Diana Franklin, and Margaret Martonosi. 2017. Programming languages and compiler design for realistic quantum hardware. *Nature* 549, 7671 (2017), 180–187.

[55] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN NOTICES* 46, 4 (2011), 53–64.

[56] Pierre Clairambault, Marc De Visme, and Glynn Winskel. 2019. Game semantics for quantum programming. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[57] John Clark and Susan Stepney. 2002. Quantum Software Engineering. In *Workshop on Grand Challenges for Computing Research*. http://www.ukcrc.org.uk/press/news/call/a5.cfm.

[58] Lori A Clarke and David S Rosenblum. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 25–37.

[59] Paul C Clements. 1996. A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design*. IEEE, 16–25.

[60] IEEE Computer Society. Software Engineering Technical Committee. 1990. IEEE standard glossary of software engineering terminology. Institute of Electrical and Electronics Engineers.

[61] Regetti Computing. 2017. Forest SDK. *Accessed on: April, 2020* (2017). http://docs.rigetti.com/en/stable/.

[62] Wikipedia Contributors. 2017. Block sphere - Wikipedia. *The Free Encyclopedia [Accessed on April 26, 2020]* (2017). https://en.wikipedia.org/wiki/Bloch_sphere

[63] Stephen A Cook and David G Mitchell. 1997. Finding hard instances of the satisfiability problem: A survey. *Satisfiability Problem: Theory and Applications* 35 (1997), 1–17.

[64] Don Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067* (2002).

[65] Rick David Craig and Stefan P Jaskiel. 2002. *Systematic software testing.* Artech house.

[66] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).

[67] Andrew W Cross, Graeme Smith, and John A Smolin. 2015. Quantum learning robust against noise. *Physical Review A* 92, 1 (2015), 012327.

[68] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. 1972. *Structured programming.* Academic Press Ltd.

[69] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2002. *Object-oriented reengineering patterns.* Elsevier.

[70] David Deutsch. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400, 1818 (1985), 97–117.

[71] David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439, 1907 (1992), 553–558.

[72] Coq Development Team. 2019. The Coq proof assistant reference manual, version 8.11. *https://coq.inria.fr/distrib/current/refman/* (2019).

[73] Ellie D'hondt and Prakash Panangaden. 2006. Quantum weakest preconditions. *Mathematical Structures in Computer Science* 16, 3 (2006), 429–451.

[74] Paul Adrien Maurice Dirac. 1981. *The principles of quantum mechanics.* Number 27. Oxford university press.

[75] Merlin Dorfman and Richard H. Thayer (eds.). 1997. Software Engineering. *IEEE Computer* (1997).

[76] Vedran Dunjko, Jacob M Taylor, and Hans J Briegel. 2016. Quantum-enhanced machine learning. *Physical review letters* 117, 13 (2016), 130501.

[77] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 574–586.

[78] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 729–739.

[79] Albert Einstein, Boris Podolsky, and Nathan Rosen. 1935. Can quantum-mechanical description of physical reality be considered complete? *Physical review* 47, 10 (1935), 777.

[80] Mark J Everitt, Michael J de C Henshaw, and Vincent M Dwyer. 2016. Quantum Systems Engineering: A structured approach to accelerating the development of a quantum technology industry. In *2016 18th International Conference on Transparent Optical Networks (ICTON)*. IEEE, 1–4.

[81] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).

[82] Samuel S Feitosa, Juliana K Vizzotto, Eduardo K Piveta, and Andre R Du Bois. 2016. FJQuantum–A Quantum Object Oriented Language. *Electronic Notes in Theoretical Computer Science* 324 (2016), 67–77.

[83] Yuan Feng, Runyao Duan, Zhengfeng Ji, and Mingsheng Ying. 2005. Semantics of a purely quantum programming language. *arXiv preprint cs/0507043* (2005).

[84] Yuan Feng, Runyao Duan, Zhengfeng Ji, and Mingsheng Ying. 2007. Proof rules for the correctness of quantum programs. *Theoretical Computer Science* 386, 1-2 (2007), 151–166.

[85] Richard P Feynman. 1982. Simulating physics with computers. *Int. J. Theor. Phys* 21, 6/7 (1982).

[86] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. 2018. Open source software in quantum computing. *arXiv preprint arXiv:1812.09167* (2018).

[87] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.

[88] Anthony Finkelsteiin and Jeff Kramer. 2000. Software engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 3–22.

[89] Aleta Berk Finnila, MA Gomez, C Sebenik, Catherine Stenson, and Jimmie D Doll. 1994. Quantum annealing: a new method for minimizing multidimensional functions. *Chemical physics letters* 219, 5-6 (1994), 343–348.

[90] RW FLOYD. 1967. Assigning meanings to program. In *Proc. Symposia in Applied Mathematics, 1967*, Vol. 19. 19–32.

[91] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification*. Springer, 65–81.

[92] Harry D Foster, Adam C Krolnik, and David J Lacey. 2006. *Assertion-Based Design.* Springer Science & Business Media.

[93] Martin Fowler. 2002. *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc.

[94] William B Frakes and Kyo Kang. 2005. Software reuse research: Status and future. *IEEE transactions on Software Engineering* 31, 7 (2005), 529–536.

[95] Buschmann Frank, Meunier Regine, Rohnert Hans, Sommerlad Peter, and Stal Michael. 1996. Pattern-oriented software architecture: a system of patterns. *Wiley, ISBN 0* 471, 95889 (1996), 7.

[96] Alfonso Fuggetta. 2000. Software process: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 25–34.

[97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns Elements of reusable object-oriented sofware.* Addison Wesley.

[98] Sunita Garhwal, Maryam Ghorani, and Amir Ahmad. 2019. Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages. *Archives of Computational Methods in Engineering* (2019), 1–22.

[99] David Garlan, Robert Monroe, and David Wile. 1997. Acme: An Architecture Description Interchange Language. *Proceedings of CASCON'97* (1997).

[100] Simon Gay. 2005. Bibliography on quantum programming languages. *Bulletin of the EATCS* 6 (2005). http://www.dcs.gla.ac.uk/~simon/quantum/qplbibD.html

[101] Simon Gay and Ian Mackie. 2010. *Semantic Techniques in Quantum Computation.* Cambridge University Press.

[102] Simon J Gay. 2006. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science* 16, 4 (2006), 581–600.

[103] Simon J Gay and Rajagopal Nagarajan. 2004. Communicating quantum processes. *Proceedings of the 2nd International Workshop on Quantum Programming Languages* (2004), 91–107.

[104] Simon J Gay and Rajagopal Nagarajan. 2005. Communicating quantum processes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 145–157.

[105] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2002. Fundamentals of Software Engineering. (2002).

[106] Craig Gidney. 2017. Visualizing 2-Qubit Entanglement. *[Accessed on: April 26, 2020]* (2017). https://algassert.com/post/1716

[107] Jean-Yves Girard. 2004. Between logic and quantic: a tract. *Linear logic in computer science* 316 (2004), 346–381.

[108] James Glanz. 1995. A quantum leap for computers? *Science* 269, 5220 (1995), 28–30.

[109] Hassan Gomaa. 2011. *Software modeling and design: UML, use cases, patterns, and software architectures.* Cambridge University Press.

[110] Daniel Gottesman. 2010. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, Vol. 68. 13–58.

[111] Daniel Gottesman and Isaac L Chuang. 1999. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature* 402, 6760 (1999), 390–393.

[112] Daniel Gottesman and Isaac L Chuang. 1999. Quantum teleportation is a universal computational primitive. *arXiv preprint quant-ph/9908010* (1999).

[113] Jonathan Grattage. 2006. *QML: A functional quantum programming language*. Ph.D. Dissertation. The University of Nottingham.

[114] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. 2013. An introduction to quantum programming in quipper. In *International Conference on Reversible Computation*. Springer, 110–124.

[115] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 333–342.

[116] Harper R Grimsley, Sophia E Economou, Edwin Barnes, and Nicholas J Mayhall. 2019. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nature communications* 10, 1 (2019), 1–9.

[117] David Gross, Yi-Kai Liu, Steven T Flammia, Stephen Becker, and Jens Eisert. 2010. Quantum state tomography via compressed sensing. *Physical review letters* 105, 15 (2010), 150401.

[118] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 212–219.

[119] Penny Grubb and Armstrong A Takang. 2003. *Software maintenance: concepts and practice*. World Scientific.

[120] Jozef Gruska. 1999. *Quantum computing*. Vol. 2005. McGraw-Hill.

[121] Gian Giacomo Guerreschi and Mikhail Smelyanskiy. 2017. Practical optimization for hybrid quantum-classical algorithms. *arXiv preprint arXiv:1701.01450* (2017).

[122] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 866–874.

[123] Ichiro Hasuo and Naohiko Hoshino. 2017. Semantics of higher-order quantum computation via geometry of interaction. *Annals of Pure and Applied Logic* 168, 2 (2017), 404–469.

[124] George T Heineman and William T Councill. 2001. Component-based software engineering. *addison-westley* (2001).

[125] Kesha Hietala. 2016. Quantum Programming Languages. (2016).

[126] Kesha Hietala, Robert Rand, and Michael Hicks. 2020. Tracking errors through types in quantum programs. In *The First International Workshop on Programming Languages for Quantum Computing (PLanQC 2020)*.

[127] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2019. Verified Optimization in a Quantum Intermediate Representation. *arXiv preprint arXiv:1904.06319* (2019).

[128] Edward Hirsch. 1985. Evolutionary acquisition of command and control systems. *Program Manager* (1985), 18–22.

[129] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[130] T. Hoare and R. Milner (eds.). 2004. Grand Challenges in Computing Research. *organised by BCS, CPHC, EPSRC, IEE, etc.* (2004). http://www.ukcrc.org.uk/press/news/report/gcresearch.cfm?type=pdf.

[131] Shahin Honarvar, Mohammadreza Mousavi, and Rajagopal Nagarajan. 2020. Property-based testing of quantum programs in Q#. In *First International Workshop on Quantum Software Engineering (Q-SE 2020)*.

[132] Yipeng Huang and Margaret Martonosi. 2018. QDB: From quantum algorithms towards correct quantum programs. *arXiv preprint arXiv:1811.05447* (2018).

[133] Yipeng Huang and Margaret Martonosi. 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*. 541–553.

[134] D-Wave Systems Inc. 2017. The D-Wave 2X quantum computer. *Accessed on: May 11, 2020* (2017). https://www.dwavesys.com/blog/2015/08/announcing-d-wave-2x-quantum-computer.

[135] Ivar Jacobson, Grady Booch, and James Rumbaugh. 1999. *The unified software development process*. Addison-wesley Reading.

[136] Sakshi Jain, Sreraman Muralidharan, and Prasanta K Panigrahi. 2009. Secure quantum conversation through non-destructive discrimination of highly entangled multipartite states. *EPL (Europhysics Letters)* 87, 6 (2009), 60008.

[137] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. 2014. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10.

[138] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. 2015. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Comput.* 45 (2015), 2–17.

[139] Xian-Min Jin, Ji-Gang Ren, Bin Yang, Zhen-Huan Yi, Fei Zhou, Xiao-Fan Xu, Shao-Kai Wang, Dong Yang, Yuan-Feng Hu, Shuo Jiang, et al. 2010. Experimental free-space quantum teleportation. *Nature photonics* 4, 6 (2010), 376–381.

[140] Philippe Jorrand. 2007. A Programmer's Survey of the Quantum Computing Paradigm. 34, 1-2 (2007).

[141] Philippe Jorrand and Marie Lalire. 2004. From quantum physics to programming languages: a process algebraic approach. In *International Workshop on Unconventional Programming Paradigms*. Springer, 1–16.

[142] Tadashi Kadowaki and Hidetoshi Nishimori. 1998. Quantum annealing in the transverse Ising model. *Physical Review E* 58, 5 (1998), 5355.

[143] Yoshihiko Kakutani. 2009. A logic for formal verification of quantum programs. In *Annual Asian Computing Science Conference*. Springer, 79–93.

[144] Peter J Karalekas, Nikolas A Tezak, Eric C Peterson, Colm A Ryan, Marcus P da Silva, and Robert S Smith. 2020. A quantum-classical cloud platform optimized for variational hybrid algorithms. *arXiv preprint arXiv:2001.04449* (2020).

[145] N Khammassi, I Ashraf, X Fu, CG Almudever, and K Bertels. 2017. QX: a high-performance quantum computer simulation platform. In *Proceedings of the Conference on Design, Automation & Test in Europe*. 464–469.

[146] Nader Khammassi, Gian G Guerreschi, Imran Ashraf, Justin W Hogaboam, Carmen G Almudever, and Koen Bertels. 2018. cqasm v1. 0: Towards a common quantum assembly language. *arXiv preprint arXiv:1805.09607* (2018).

[147] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. 2018. Strawberry Fields: A Software Platform for Photonic Quantum Computing. *arXiv preprint arXiv:1804.03159* (2018).

[148] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. 2019. Strawberry fields: A software platform for photonic quantum computing. *Quantum* 3 (2019), 129.

[149] Alexei Yu Kitaev, Alexander Shen, Mikhail N Vyalyi, and Mikhail N Vyalyi. 2002. *Classical and quantum computation*. Number 47. American Mathematical Soc.

[150] Andreas Klappenecker and Martin Rötteler. 2003. Quantum software reusability. *International Journal of Foundations of Computer Science* 14, 05 (2003), 777–796.

[151] Claire Knight and Malcolm Munro. 1999. Comprehension with [in] virtual environment visualisations. In *Proceedings Seventh International Workshop on Program Comprehension*. IEEE, 4–11.

[152] Emmanuel Knill. 1996. *Conventions for quantum pseudocode*. Technical Report. Los Alamos National Lab., NM (United States).

[153] Emanuel Knill, Raymond Laflamme, and Wojciech H Zurek. 1998. Resilient quantum computation: error models and thresholds. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454, 1969 (1998), 365–384.

[154] EH Knill and MA Nielsen. 2000. Encyclopedia of Mathematics, Supplement III, chapter Theory of quantum computation. (2000).

[155] Andrew Ko and Brad Myers. 2008. Debugging reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 301–310.

[156] Wojtek Kozaczynski and Grady Booch. 1998. Component-based software engineering. *IEEE software* 15, 5 (1998), 34.

[157] Sebastian Krämer, David Plankensteiner, Laurin Ostermann, and Helmut Ritsch. 2018. QuantumOptics. jl: A Julia framework for simulating open quantum systems. *Computer Physics Communications* 227 (2018), 109–116.

[158] Charles W Krueger. 1992. Software reuse. *Comput. Surveys* 24, 2 (1992), 131–183.

[159] Tom Krüger and Wolfgang Mauerer. 2020. Quantum Annealing-Based Software Components: An Experimental Case Study with SAT Solving. In *International Workshop on the Quantum Software Engineering & Programming*.

[160] Marie Lalire and Philippe Jorrand. 2004. A process algebraic approach to concurrent and distributed quantum computation: operational semantics. *Proceedings of the 2nd International Workshop on Quantum Programming Languages* (2004), 109–126.

[161] Benjamin P Lanyon, James D Whitfield, Geoff G Gillett, Michael E Goggin, Marcelo P Almeida, Ivan Kassal, Jacob D Biamonte, Masoud Mohseni, Ben J Powell, Marco Barbieri, et al. 2010. Towards quantum chemistry on a quantum computer. *Nature chemistry* 2, 2 (2010), 106–111.

[162] Andrei Lapets, Marcus P da Silva, Mike Thome, Aaron Adler, Jacob Beal, and Martin Rötteler. 2013. QuaFL: a typed DSL for quantum programming. In *Proceedings of the 1st annual workshop on Functional programming concepts in domain-specific languages*. 19–26.

[163] Ryan LaRose. 2019. Overview and comparison of gate level quantum software platforms. *Quantum* 3 (2019), 130.

[164] Beatrice Lazzerini and Lanfranco Lopriore. 1992. *Program Debugging Environments, Design and Utilization*. Ellis Horwood/Simon & Schuster.

[165] Raimondas Lencevicius. 2000. *Advanced debugging methods*. Vol. 568. Springer Science & Business Media.

[166] Frank Leymann. 2019. Towards a Pattern Language for Quantum Algorithms. In *International Workshop on Quantum Technology and Optimization Problems*. Springer, 218–230.

[167] Frank Leymann, Johanna Barzen, Michael Falkenthal, Daniel Vietz, Benjamin Weder, and Karoline Wild. 2020. Quantum in the Cloud: Application Potentials and Research Opportunities. *arXiv preprint arXiv:2003.06256* (2020).

[168] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2019. Poq: Projection-based Runtime Assertions for Debugging on a Quantum Computer. *arXiv preprint arXiv:1911.12855* (2019).

[169] Yangjia Li and Mingsheng Ying. 2014. Debugging quantum processes using monitoring measurements. *Physical Review A* 89, 4 (2014), 042338.

[170] Bennett P Lientz and E Burton Swanson. 1980. *Software maintenance management*. Addison-Wesley Longman Publishing Co., Inc.

[171] Richard J Lipton and Kenneth W Regan. 2014. *Quantum algorithms via linear algebra: a primer.* MIT Press.

[172] Ji Liu, Gregory T Byrd, and Huiyang Zhou. 2020. Quantum circuits for dynamic runtime assertions in quantum computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 1017–1030.

[173] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal verification of quantum algorithms using quantum Hoare logic. In *International conference on computer aided verification.* Springer, 187–207.

[174] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. 2018. $Q|SI\rangle$: A Quantum Programming Environment. In *Symposium on Real-Time and Hybrid Systems.* Springer, 133–164.

[175] Xiao Liu and John Kubiatowicz. 2013. Chisel-Q: Designing quantum circuits with a scala embedded language. In *2013 IEEE 31st International Conference on Computer Design (ICCD).* IEEE, 427–434.

[176] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. 1995. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 21, 4 (1995), 336–354.

[177] Jeff Magee and Jeff Kramer. 1996. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes* 21, 6 (1996), 3–14.

[178] Margaret Martonosi and Martin Roetteler. 2019. Next Steps in Quantum Computing: Computer Science's Role. *arXiv preprint arXiv:1903.10541* (2019).

[179] Shaw Mary and Garlan David. 1996. Software architecture: perspectives on an emerging discipline. *Prentice-Hall* (1996).

[180] Wolfgang Mauerer. 2005. Semantics and simulation of communication in quantum programming. *arXiv preprint quant-ph/0511145* (2005).

[181] Philip Maymin. 1996. Extending the lambda calculus to express randomized and quantumized algorithms. *arXiv preprint quant-ph/9612052* (1996).

[182] Sam McArdle, Suguru Endo, Alan Aspuru-Guzik, Simon Benjamin, and Xiao Yuan. 2018. Quantum computational chemistry. *arXiv preprint arXiv:1808.10402* (2018).

[183] Hefedh Mili, Ali Mili, Sherif Yacoub, and Edward Addy. 2001. *Reuse-based software engineering: techniques, organization, and controls.* Wiley-Interscience.

[184] Andriy Miranskyy and Lei Zhang. 2019. On testing quantum programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER).* IEEE, 57–60.

[185] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. 2020. Is Your Quantum Program Bug-Free? *arXiv preprint arXiv:2001.10870* (2020).

[186] Jaroslaw Miszczak. 2011. Models of quantum computation and quantum programming languages. *Bulletin of the Polish Academy of Sciences: Technical Sciences* 59, 3 (2011), 305–324.

[187] Jarosław Adam Miszczak. 2012. *High-level structures for quantum computing.* Vol. 4. Morgan & Claypool Publishers. 1–129 pages.

[188] Hynek MlnařÍk. 2006. Introduction to LanQ–an Imperative Quantum Programming Language. (2006).

[189] Hynek MlnařÍk. 2007. Operational semantics and type soundness of quantum programming language LanQ. *arXiv preprint arXiv:0708.0890* (2007).

[190] Hynek MlnařÍk. 2008. Semantics of quantum programming language LanQ. *International Journal of Quantum Information* 6, supp01 (2008), 733–738.

[191] Hynek Mlnarık. September, 2007. *Quantum programming language LanQ.* Ph.D. Dissertation. Masaryk University, Faculty of Informatics.

[192] Ashley Montanaro. 2016. Quantum algorithms: an overview. *npj Quantum Information* 2, 1 (2016), 1–8.

[193] Michele Mosca. 2008. Quantum algorithms. *arXiv preprint arXiv:0808.0369* (2008).

[194] Michele Mosca. 2018. Cybersecurity in an era with quantum computers: will we be ready? *IEEE Security & Privacy* 16, 5 (2018), 38–41.

[195] Michele Mosca, Martin Roetteler, and Peter Selinger. 2019. Quantum Programming Languages (Dagstuhl Seminar 18381). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[196] Glenford J Myers. 1979. *The art of software testing.* John Wiley & Sons.

[197] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing.* John Wiley & Sons.

[198] Mikio Nakahara and Tetsuo Ohmi. 2008. *Quantum computing: from linear algebra to physical realizations.* CRC press.

[199] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *Journal of Functional Programming* 18, 5-6 (2008), 865–911.

[200] Engineering National Academies of Sciences, Medicine, et al. 2019. *Quantum computing: progress and prospects.* National Academies Press.

[201] Peter Naur and Brian Randell. 1969. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968. (1969).

[202] Michael A Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information. (2002).

[203] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis.* Springer.

[204] Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, and Alán Aspuru-Guzik. 2017. Quantum information and computation for chemistry. *arXiv preprint arXiv:1706.05413* (2017).

[205] Peter JJ O'Malley, Ryan Babbush, Ian D Kivlichan, Jonathan Romero, Jarrod R McClean, Rami Barends, Julian Kelly, Pedram Roushan, Andrew Tranter, Nan Ding, et al. 2016. Scalable quantum simulation of molecular energies. *Physical Review X* 6, 3 (2016), 031007.

[206] Bernhard Ömer. 1998. *A procedural formalism for quantum computing.* Master's thesis. Department of Theoretical Physics, Technical University of Vienna.

[207] Bernhard Ömer. 2000. *Quantum programming in QCL.* Master's thesis. Institute of Information Systems, Technical University of Vienna.

[208] Bernhard Ömer. 2003. *Structured quantum programming.* Ph.D. Dissertation. Institute for Theoretical Physics, Technical University of Vienna.

[209] Bernhard Ömer. 2005. Classical concepts in quantum programming. *International Journal of Theoretical Physics* 44, 7 (2005), 943–955.

[210] Luis Ortiz-Gutiérrez, Bruna Gabrielly, Luis F Muñoz, Kainã T Pereira, Jefferson G Filgueiras, and Alessandro S Villar. 2017. Continuous variables quantum computation over the vibrational modes of a single trapped ion. *Optics Communications* 397 (2017), 166–174.

[211] Michele Pagani, Peter Selinger, and Benoît Valiron. 2014. Applying quantitative semantics to higher-order quantum computing. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 647–658.

[212] Scott Pakin. 2016. A quantum macro assembler. In *2016 IEEE High Performance Extreme Computing Conference (HPEC).* IEEE, 1–8.

[213] Luca Paolini, Mauro Piccolo, and Margherita Zorzi. 2019. QPCF: Higher-Order Languages and Quantum Circuits. *Journal of Automated Reasoning* 63, 4 (2019), 941–966.

[214] Luca Paolini, Luca Roversi, and Margherita Zorzi. 2019. Quantum programming made easy. In *2018 Joint International Workshop on Linearity and Trends in Linear Logic and Applications, Linearity-TLLA 2018,* Vol. 292. 133–147.

[215] Luca Paolini and Margherita Zorzi. 2017. qPCF: A Language for Quantum Circuit Computations. In *International Conference on Theory and Applications of Models of Computation.* Springer, 455–469.

[216] Microsoft Patterns. 2009. *Microsoft application architecture guide.* Microsoft Press.

[217] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. *ACM SIGPLAN Notices* 52, 1 (2017), 846–858.

[218] Ricardo Pérez-Castillo. 2020. Reengineering of Information Systems toward Classical-Quantum Systems. In *International Workshop on the Quantum Software Engineering & Programming.*

[219] Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman, and Mario Piattini. 2011. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces* 33, 6 (2011), 519–532.

[220] Carlos A. Pérez-Delgado and Hector G. Perez-Gonzalez. 2020. Towards a Quantum Software Modeling Language. In *First International Workshop on Quantum Software Engineering (Q-SE 2020).*

[221] Marek Perkowski, Martin Lukac, Pawel Kerntopf, Mikhail Pivtoraiko, Michele Folgheraiter, Yong Woo Choi, Jung-wook Kim, Dongsoo Lee, Woong Hwangbo, and Hyungock Kim. 2003. A hierarchical approach to computer-aided design of quantum circuits. In *6th International Symposium on Representations and Methodology of Future Computing Technologies.* 201–209.

[222] Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes* 17, 4 (1992), 40–52.

[223] Mario Piattini, Guido Peterssen, Ricardo Pérez-Castillo, Jose Luis Hevia, Manuel A. Serrano, Guillermo Hernández, Ignacio García Rodríguez de Guzmán, Claudio Andrés Paradela, Macario Polo, Ezequiel Murina, Luis Jiménez, Juan Carlos Marqueño, Ramsés Gallego, Jordi Tura, Frank Phillipson, Juan M. Murillo, Alfonso Niño, and Moisés Rodríguez. 2020. The Talavera Manifesto for Quantum Software Engineering and Programming. In *Short Papers Proceedings of the 1st International Workshop on the QuANtum SoftWare Engineering & pRogramming, Talavera de la Reina, Spain, February 11-12, 2020 (CEUR Workshop Proceedings),* Mario Piattini, Guido Peterssen, Ricardo Pérez-Castillo, Jose Luis Hevia, and Manuel A. Serrano (Eds.), Vol. 2561. CEUR-WS.org, 1–5. http://ceur-ws.org/Vol-2561/paper0.pdf

[224] John Preskill. 1999. Plug-in quantum software. *Nature* 402, 6760 (1999), 357–358.

[225] John Preskill. 2018. Lecture notes for physics 229: Quantum information and computation. *California Institute of Technology* 16 (2018). Availablefromhttp://www.theory.caltech.edu/people/preskill/ph229/#lecture

[226] Roger S Pressman. 2010. *Software engineering: A Practitioner's approach.* McGraw-Hill.

[227] Matt Purkeypile. 2009. *Cove: A practical quantum computer programming framework.* Ph.D. Dissertation. Colorado Technical University. https://arxiv.org/abs/0911.2423

[228] Robert Rand. 2018. *Formally verified quantum programming*. Ph.D. Dissertation. University of Pennsylvania.

[229] Robert Rand, Kesha Hietala, and Michael Hicks. 2019. Formal Verification vs. Quantum Uncertainty. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[230] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. 2014. Quantum support vector machine for big data classification. *Physical review letters* 113, 13 (2014), 130503.

[231] Markus Reiher, Nathan Wiebe, Krysta M Svore, Dave Wecker, and Matthias Troyer. 2017. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences* 114, 29 (2017), 7555–7560.

[232] IBM Research. 2017. Qiskit. *Accessed on: April, 2020* (2017). https://qiskit.org

[233] IBM Research. 2020. Entanglion. *[Accessed on: April 26, 2020]* (2020). https://entanglion.github.io/

[234] Francisco Rios and Peter Selinger. 2017. A categorical model for a quantum circuit description language. In *Proceedings 14th International Conference on Quantum Physics and Logic (QPL 2017)*. 164–178.

[235] J ALLEN Robert. 1997. *A formal approach to software architecture*. Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University.

[236] Martin Roetteler, Krysta M Svore, Dave Wecker, and Nathan Wiebe. 2017. Design automation for quantum architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1312–1317.

[237] Neil Julien Ross. 2015. *Algebraic and Logical Methods in Quantum Computation*. Ph.D. Dissertation. Department of Mathematics and Statistics, Dalhousie University, 2015. Available from arXiv:1510.02198.

[238] Winston W Royce. August, 1970. Managing the development of large software systems: concepts and techniques. In *Proceedings of the WESCON*.

[239] Roland Rüdiger. 2007. Quantum Programming Languages: An Introductory Overview. *Comput. J.* 50, 2 (2007), 134–150.

[240] James Rurnbaugh, Ivar Jacobson, and Grady Booch. 1999. *The Unified Modeling Language Reference Manual*. addison-wesley.

[241] Jeff W Sanders and Paolo Zuliani. 2000. Quantum programming. In *International Conference on Mathematics of Program Construction*. Springer, 80–99.

[242] Erwin Schrödinger. 1935. Discussion of probability relations between separated systems. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 31. Cambridge University Press, 555–563.

[243] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. 2003. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional.

[244] Peter Selinger. 2004. A brief survey of quantum programming languages. In *International Symposium on Functional and Logic Programming*. Springer, 1–6.

[245] Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586.

[246] Peter Selinger. 2004. Towards a semantics for higher-order quantum computation. In *Proceedings of the 2nd International Workshop on Quantum Programming Languages, TUCS General Publication No*, Vol. 33. 127–143.

[247] Peter Selinger and Benoit Valiron. 2006. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science* 16, 3 (2006), 527–552.

[248] Kostya Serebryany. 2015. libFuzzer–a library for coverage-guided fuzz testing. *LLVM project* (2015).

[249] Changpeng Shao, Yang Li, and Hongbo Li. 2019. Quantum algorithm design: Techniques and applications. *Journal of Systems Science and Complexity* 32, 1 (2019), 375–452.

[250] Ehud Y Shapiro. 1982. Algorithmic program diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 299–308.

[251] Ehud Y Shapiro. 1983. Algorithmic Program Debugging. (1983).

[252] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. 1995. Abstractions for software architecture and tools to support them. *IEEE transactions on software engineering* 21, 4 (1995), 314–335.

[253] Ruslan Shaydulin, Caleb Thomas, and Paige Rodeghero. 2020. Making Quantum Computing Open: Lessons from Open-Source Projects. In *First International Workshop on Quantum Software Engineering (Q-SE 2020)*.

[254] Seung Woo Shin, Graeme Smith, John A Smolin, and Umesh Vazirani. 2014. How" quantum" is the D-Wave machine? *arXiv preprint arXiv:1401.7087* (2014).

[255] Peter W Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 124–134.

[256] Peter W Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Physical review A* 52, 4 (1995), R2493.

[257] Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review* 41, 2 (1999), 303–332.

[258] Kartik Singhal. 2019. Hoare types for quantum programming languages. *POPL 2020 Student Research Competition Extended Abstract* (2019). http://ks.cs.uchicago.edu/publication/quantum-hoare-types-src/Quantum-Hoare-Types-SRC.pdf

[259] Robert S Smith, Michael J Curtis, and William J Zeng. 2016. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355* (2016).

[260] Balwinder Sodhi. 2018. Quality Attributes on Quantum Computing Platforms. *arXiv preprint arXiv:1803.07407* (2018).

[261] Donald A Sofge. 2008. A survey of quantum programming languages: History, methods, and tools. In *Second International Conference on Quantum, Nano and Micro Technologies (ICQNM 2008)*. IEEE, 66–71.

[262] Ian Sommerville. 2011. Software engineering 9th Edition. *ISBN-10* 137035152 (2011), 18.

[263] Milan Spišiak and Ján Kollár. 2017. Quantum programming: A review. In *2017 IEEE 14th International Scientific Conference on Informatics*. IEEE, 353–358.

[264] John Stasko, John Domingue, Marc H Brown, and Blaine Price. 1998. Software Visualization: Programming as a Multimedia Experience. (1998).

[265] Andrew M Steane. 1996. Error correcting codes in quantum theory. *Physical Review Letters* 77, 5 (1996), 793.

[266] Damian S Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (2018), 49.

[267] Susan Stepney, Samuel L Braunstein, John A Clark, Andy Tyrrell, Andrew Adamatzky, Robert E Smith, Tom Addis, Colin Johnson, Jonathan Timmis, Peter Welch, et al. 2005. Journeys in non-classical computation I: A grand challenge for computing research. *International Journal of Parallel, Emergent and Distributed Systems* 20, 1 (2005), 5–19.

[268] Susan Stepney, Samuel L Braunstein, John A Clark, Andy Tyrrell, Andrew Adamatzky, Robert E Smith, Tom Addis, Colin Johnson, Jonathan Timmis, Peter Welch, et al. 2006. Journeys in non-classical computation II: initial journeys and way points. *The International Journal of Parallel, Emergent and Distributed Systems* 21, 2 (2006), 97–125.

[269] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. 1–10.

[270] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance*. Artech House.

[271] Google AI Quantum team. 2018. Cirq. (2018). https://github.com/quantumlib/Cirq

[272] ProjectQ Team. 2017. ProjectQ. *Accessed on: April, 2020* (2017). https://projectq.ch/

[273] Q# Team. 2017. Testing and debugging. *Accessed on: April, 2020* (2017). https://docs.microsoft.com/en-us/quantum/techniques/testing-and-debugging?view=qsharp-preview%5C&tabs=tabid-vs2019

[274] Jayne Thompson, Kavan Modi, Vlatko Vedral, and Mile Gu. 2018. Quantum plug n'play: modular computation in the quantum regime. *New Journal of Physics* 20, 1 (2018), 013004.

[275] Jordi Tura. 2020. Quantum algorithms for near-term devices. In *International Workshop on the Quantum Software Engineering & Programming*.

[276] Dominique Unruh. 2006. Quantum programming languages. *Informatik-forschung und entwicklung* 21, 1-2 (2006), 55–63.

[277] Dominique Unruh. 2019. Quantum relational Hoare logic. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.

[278] Macario Polo Usaola. 2020. Quantum Software Testing. In *International Workshop on the Quantum Software Engineering & Programming*.

[279] Mark Utting and Bruno Legeard. 2010. *Practical model-based testing: a tools approach*. Elsevier.

[280] Benoît Valiron. 2013. Quantum computation: From a programmer's perspective. *New Generation Computing* 31, 1 (2013), 1–26.

[281] Benoît Valiron, Neil J Ross, Peter Selinger, D Scott Alexander, and Jonathan M Smith. 2015. Programming the quantum future. *Commun. ACM* 58, 8 (2015), 52–61.

[282] AJ Van Tonder. 2003. *Quantum computation, categorical semantics and linear logic*. Technical Report.

[283] André Van Tonder. 2004. A lambda calculus for quantum computation. *SIAM J. Comput.* 33, 5 (2004), 1109–1135.

[284] Lieven Vandenberghe and Stephen Boyd. 1996. Semidefinite programming. *SIAM review* 38, 1 (1996), 49–95.

[285] Edmond VanDoren. 1997. Maintenance of operational systems–an overview. *Software technology Roadmap. Carnegie Mellon Software Engineering Institute* (1997).

[286] Juliana Kaizer Vizzotto and Antônio Carlos da Rocha Costa. 2005. Concurrent quantum programming in Haskell. In *VII Congresso Brasileiro de Redes Neurais, Sessao de Computaçao Quântica*. Citeseer.

[287] Markus Völter. 2006. Software Architecture-A pattern language for building sustainable software architectures.. In *EuroPLoP*. 31–66.

[288] Di Wang and Robert Kleinberg. 2009. Analyzing quadratic unconstrained binary optimization problems via multicommodity flows. *Discrete Applied Mathematics* 157, 18 (2009), 3746–3753.

[289] Jiyuan Wang, Ming Gao, Yu Jiang, Jianguang Lou, Yue Gao, Dongmei Zhang, and Jiaguang Sun. 2018. QuanFuzz: Fuzz Testing of Quantum Program. *arXiv*

preprint arXiv:1810.10310 (2018).

[290] Dave Wecker, Bela Bauer, Bryan K Clark, Matthew B Hastings, and Matthias Troyer. 2013. Can quantum chemistry be performed on a small quantum computer. *arXiv preprint arXiv:1312.1695* (2013), 15.

[291] Dave Wecker, Bela Bauer, Bryan K Clark, Matthew B Hastings, and Matthias Troyer. 2014. Gate-count estimates for performing quantum chemistry on small quantum computers. *Physical Review A* 90, 2 (2014), 022305.

[292] Dave Wecker and Krysta M Svore. 2014. LIQUi|⟩: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467* (2014).

[293] Colin P Williams and Alexander G Gray. 1998. Automated design of quantum circuits. In *NASA International Conference on Quantum Computing and Quantum Communications*. Springer, 113–125.

[294] HM Wiseman and GJ Milburn. 1993. Interpretation of quantum jump and diffusion processes illustrated on the Bloch sphere. *Physical Review A* 47, 3 (1993), 1652.

[295] Marty J Wolf, Frances S Grodzinsky, and Keith W Miller. 2012. Artificial agents, cloud computing, and quantum computing: Applying FloridiâĂŽs method of levels of abstraction. In *Luciano FloridiâĂŽs Philosophy of Technology*. Springer, 23–41.

[296] Jim Woodcock and Jim Davies. 1996. Using Z: Specification, Refinement, and Proof. (1996).

[297] www.merriam webster.com. 2020. Software Engineering. *Accessed on: June 28, 2020* (2020). https://qiskit.org

[298] Jia-Fu Xu, Fang-Min Song, Shi-Jun Qian, Jing-An Dai, and Yun-Jie Zhang. 2008. Quantum programming language NDQJava. *Journal of Software* 19, 1 (2008), 1–8.

[299] Zhenyu Yang, James Z Fan, Andrew H Proppe, F Pelayo García de Arquer, David Rossouw, Oleksandr Voznyy, Xinzheng Lan, Min Liu, Grant Walters, Rafael Quintero-Bermudez, et al. 2017. Mixed-quantum-dot solar cells. *Nature communications* 8, 1 (2017), 1–9.

[300] Mingsheng Ying. 2012. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2012), 1–49.

[301] Mingsheng Ying. 2016. *Foundations of Quantum Programming*. Morgan Kaufmann.

[302] Mingsheng Ying, Yuan Feng, Runyao Duan, Yangjia Li, and Nengkun Yu. 2012. Quantum programming: From theories to implementations. *Chinese science bulletin* 57, 16 (2012), 1903–1909.

[303] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. 2017. Invariants of quantum programs: characterisations and generation. *ACM SIGPLAN Notices* 52, 1 (2017), 818–832.

[304] Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. 2013. Verification of quantum programs. *Science of Computer Programming* 78, 9 (2013), 1679–1700.

[305] Edward Yourdon and Larry L Constantine. 1979. *Structured design: fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc.

[306] Michal Zalewski. 2007. American fuzzy lop.(2007). (2007).

[307] Christof Zalka. 1998. Efficient simulation of quantum systems by quantum computers. *Fortschritte der Physik: Progress of Physics* 46, 6-8 (1998), 877–879.

[308] Anton Zeilinger. 1999. Experiment and the foundations of quantum physics. In *More Things in Heaven and Earth*. Springer, 482–498.

[309] Huiyang Zhou and Gregory T Byrd. 2019. Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation. *IEEE Computer Architecture Letters* 18, 2 (2019), 111–114.

[310] Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1149–1162.

[311] Margherita Zorzi. 2019. Quantum Calculi - From Theory to Language Design. *Applied Sciences* 9, 24 (2019), 5472.

[312] Paolo Zuliani. 2001. *Formal reasoning for quantum mechanical nonlocality*. Technical Report. Technical Report RR-01-05, Oxford University Computing Laboratory.

[313] Paolo Zuliani. 2001. *Quantum programming*. Ph.D. Dissertation. University of Oxford.

[314] Paolo Zuliani. 2004. Non-deterministic quantum programming. *Proc. QPL* (2004), 179–195.